44680

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Division
Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

JAMES O. ACHUGBUE

Date of Birth — Date de naissance

JAN 13, 1947

Country of Birth — Lieu de naissance

NIGERIA

Permanent Address — Résidence fixe

ARAGBA POSTAL - AGENCY,
VIA ABRAKA P.O.,
BENDEL STATE, NIGERIA

Title of Thesis — Titre de la thèse

THE COMPLEXITY OF SOME DETERMINISTIC SCHEDULING PROBLEMS.

University — Université

UNIVERSITY OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

Ph. D.

Year this degree conferred — Année d'obtention de ce grade

1980

Name of Supervisor — Nom du directeur de thèse

DR. FRANCIS Y. CHIN

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date

22nd April, 1980

Signature

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Ottawa, Canada
K1A 0N4

NL-339 (Rev. 8/80)

THE UNIVERSITY OF ALBERTA

THE COMPLEXITY OF SOME DETERMINISTIC SCHEDULING PROBLEMS

by

© JAMES O. ACHUGBUE

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1980

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "The Complexity of Some Deterministic Scheduling Problems" submitted by James O. Achugbue in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Supervisor

External Examiner

DATE March 27 1980

To my wife, Molly,

my son, Ani, and daughter, Elsie

ABSTRACT

This report presents several significant results on deterministic processor scheduling. For some minimal length problems, polynomial algorithms are given; namely, an $O(n^6)$ algorithm for two special types of three-processor flow shops and an $O(n^2)$ algorithm for an m-processor bound system with equal execution time tasks on two task chains. Using a dynamic programming approach, an $O(n^2 I)$ algorithm is also outlined for a tree-structured set of equal execution time tasks on a 2-processor bound system, where I is the number of terminal subsets of the tree. This algorithm is not polynomial in n but is a significant improvement over the alternative of simple enumeration.

Several problems are also shown to be NP-complete. These are minimizing schedule length on two-procesor bound unit execution time task systems even when the precedence constraints consist of chains only, 2-maximal three-processor flow shops and 1 or 3 maximal/minimal flow shops, and minimizing the mean flow time on the two-processor open shop. With the exception of the 2-maximal flow shop, the results are strong NP-complete reductions to the 3-PARTITION problem.

Finally, in the area of performance bounds, tight bounds are obtained on the lengths of list schedules on

identical processors for independent tasks with similar execution times, and on the mean flow times of arbitrary and SPT schedules for the open shop.

ACKNOWLEDGEMENTS

I offer sincere thanks to my supervisor, Dr. Francis Chin, for his guidance and assistance during the course of this project, and to the members of my committee, Drs. Mary McLeish, H. Abbot, Len Schubert, and T. C. Hu, for their very helpful suggestions.

I am grateful to the Canadian Commonwealth Scholarship and Fellowship Committee and the Department of Computing Science for financial assistance, without which this project would not have been possible.

Above all, special thanks go to my wife for her support and understanding, my uncle, Augustine Adjedjevbe, for his early guidance and continued interest in my education, and my close friend, Sunday Achuba, for invaluable assistance in recent years.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 The Scheduling Function	2
	1.2 Scheduling Theory	3
	1.2.1 The General Model and Basic Notation	4
	1.2.2 Problem Classification and Reduction	5
	1.2.3 Solution Techniques	9
	1.3 Outline of Thesis	11
2	BOUNDS ON SCHEDULES FOR INDEPENDENT TASKS	13
	2.1 Survey	13
	2.2 Normalization	16
	2.3 Bounds for Similar Tasks	30
	2.3.1 Tasks with Largest Execution Time Ratio ≤ 3	30
	2.3.2 Tasks with Largest Execution Time Ratio ≤ 2	47
	2.4 Discussion	53
3	PROCESSOR BOUND SYSTEMS	56
	3.1 Survey	57
	3.2 Definitions	58
	3.3 Complexity of the k-Chain Problem	59
	3.4 Solution of the 2-Chain Problem	62
	3.5 Extension to More Complex Precedence Graphs	67
	3.6 Discussion	83

TABLE OF CONTENTS (continued)

Chapter		Page
4	FLOW SHOP SCHEDULES	85
	4.1 Survey	86
	4.2 J-Maximal and J-Minimal Flow Shops	88
	4.2.1 2-Minimal Flow Shop: A Solvable Case	90
	4.2.2 NP-Complete Cases	92
	4.3 Ordered Three Stage Flow Shops	103
	4.3.1 LMS Flow Shops	104
	4.3.2 SML Flow Shops	124
	4.4 Discussion	125
5	OPEN SHOP SCHEDULES	127
	5.1 Survey	127
	5.2 Complexity of Mean Flow Schedules	128
	5.3 Heuristic Solutions	148
	5.4 Discussion	154
6	CONCLUDING REMARKS	155
	BIBLIOGRAPHY	158

LIST OF FIGURES

Figure	Description	Page
1	Schedules for Lemma 2.1	18
2	Structures of List Schedule and Optimal Schedule, Section 2.2	19
3	Operation II of Lemma 2.7	27
4	Order of Tasks in Normalized Counter-Example, r≤3, Section 2.2	32
5	Detail at Deletion Area of Figure 4	33
6	Examples which Achieve the Bounds of Theorem 2.1	39
7	Possible Optimal Schedules, m=4, r≤3, Theorem 2.2	41
8	Worst Case for m=4, r≤3, Theorem 2.2	43
9	Possible Optimal Schedules, m=5, r≤3, Theorem 2.2	44
10	Worst Case for m=5, r≤3, Theorem 2.2	46
11	Worst Cases for m=2,3 and even m>3, r≤2, Theorems 2.3, 2.4	49
12	Order of Tasks in Normalized Counter-Example, r≤2, Theorem 2.5	52
13	Worst Cases for Odd m>4, r≤2, Theorem 2.5	54
14	Worst Ratios Versus Similarity r, Section 2.4	55
15	Template of Theorem 3.1	61
16	Optimal Schedule of Example 3.1	66
17	Tree of Example 3.2	75
18	Optimal Schedule of Theorem 4.2	93
19	Unique Optimal Permutation Schedule for n=4, Lemma 4.1	95

LIST OF FIGURES (continued)

Figure	Description	Page
20	Unique Optimal Permutation Schedule for n=3, Lemma 4.2	98
21	Relationships and Complexities of the Special Three-stage Flow Shops	126
22	Skeleton of Optimal Schedule of Theorem 5.1, n=4	131
23	Schedule Detail for Lemma 5.4	141
24	Schedules for Example 5.1	151
25	Schedules for Example 5.2	153

LIST OF PROCEDURES

Procedure		Description	Page
1	CHAIN-ALG:	Algorithm for the 2-chain problem, Section 3.4	64
2	LABELTREE:	Labelling stage of algorithm for the tree problem, Section 3.5	71
3	DECODE:	Decoding stage of algorithm for the tree problem, Section 3.5	73
4	TREE-ALG:	Solution of the tree problem, Section 3.5	82
5	LENGTH:	Procedure for updating lengths of partial permutation schedules, Section 4.3.1	116
6	CHECK:	Procedure for checking validity of a permutation, Section 4.3.1	117
7	CHECKCYCLE:	procedure for checking and modifying permutations, Section 4.3.1	118
8	ALSORT:	Search procedure for eligible almost sorted permutations, Section 4.3.1	119
9	LMS-FLOW:	LMS flow shop algorithm, main procedure, Section 4.3.1	123

Chapter One

INTRODUCTION

The scheduling problem is concerned with the allocation of resources over time to perform a collection of tasks. It is deterministic when the information describing the tasks is assumed to be known in advance; that is, the task system is static as opposed to dynamic systems in which new tasks may be added while scheduling is in progress. The problem may be broken down into that of allocation and sequencing. In other words, given a set of processors and tasks, a schedule is fully specified when the tasks to be performed on each processor are determined (allocation) and the order in which the tasks on each processor will be done is given (sequencing) subject to prespecified constraints. Occasionally, however, only one of these two elements may be present.

Scheduling may be viewed, firstly, as a decision-making function and, secondly, as a body of theory. This thesis is concerned mainly with the task of furthering understanding of the scheduling problem from the theoretical viewpoint. In this chapter, the scheduling function and the theoretical problem are briefly discussed and, finally, an outline of the remaining chapters is presented.

1.1 The Scheduling Function

As a decision-making function, scheduling is of general practical value; one hardly need dwell on motivations for its study. Its applicability ranges from the simple, such as organizing an effective workday or preparing a meal, to the most complex, such as planning the operations for a large computer installation or job-shop environment.

An important element of the scheduling function is that of evaluating a set of alternative courses of action and selecting the cheapest depending on stated objectives. In general, the measurement of the costs in a system due to scheduling decisions is a difficult task. However, three measures of performance have become prevalent, namely, efficient utilization of resources, rapid response to demands, and close conformance to prescribed deadlines. These are closely modelled when schedules are designed to minimize the time taken to finish all the tasks, to minimize the mean time tasks spend in the system, and to minimize lateness or tardiness. Thus, the mathematical models with which scheduling theory is concerned are of value in their ability to represent the general structure and essential properties of real life scheduling problems.

1.2 Scheduling Theory

Scheduling models come in several variations depending upon the original problems from which the models are abstracted. The general model given below captures the essence of most of these models and serves to present the basic notation which will be employed subsequently. Variations which are covered in subsequent chapters will be noted as they are encountered.

From the theoretical viewpoint, scheduling problems are problems of combinatorial optimization. Consequently, several well known approaches (Aho, Hopcroft & Ullman, 1974; Baker, 1974; Lawler, 1976; Weide, 1977; etc) for studying such problems are applicable. Conversely, advances in scheduling theory tend to have similar effects on other problems of combinatorial optimization. After the presentation of the general model, this section is concluded with a discussion of problem classification and reduction, the techniques for obtaining efficient algorithms, and heuristic approaches. The ideas are applicable to any combinatorial optimization problem but the discussion is centered mainly on their application to scheduling problems.

1.2.1 The General Model and Basic Notation

The model has the following constituents; (a) resources, (b) task system, (c) sequencing constraints and (d) the performance measure to be applied.

(a) The resources consist of a set of m processors $\{P_1,\ldots,P_m\}$. In the most general model there is also a set of additional resource types, but this will not be covered in this thesis.

(b) The task system can be defined as the system $(\{T_i\}, <, \{t_i[j]\}, \{W_i\})$ as follows:

(1) $\{T_i\}$, $1 \leq i \leq n$, is a set of n tasks to be executed.

(2) $<$ is an (irreflexive) partial order defined on the set of tasks which specifies operational precedence constraints. That is, $T_i < T_j$ implies that T_i must be completed before the execution of T_j can begin.

(3) $t_i[j] \geq 0$ is the time required to execute task T_i on processor P_j, $1 \leq i \leq n$, $1 \leq j \leq m$. When the execution time is independent of the processor, the second index, j, is dropped giving time t_i for task T_i.

(4) The weights W_i, $1 \leq i \leq n$, are interpreted as deferral cost rates and are assumed to be constant. Thus, the cost of finishing task T_i at time t is simply $W_i t$.

(c) Schedules may be <u>preemptive</u> or <u>nonpreemptive</u>. In a nonpreemptive schedule, a task cannot be interrupted once it has begun execution. A preemptive schedule allows a task to

be interrupted and removed from the processor under the assumption that it will eventually receive all of its required execution time, and there is no loss of execution time due to preemptions.

(d) For any task T_i, the time at which the execution of T_i is started will be denoted by $S(T_i)$ and the time at which T_i is completed will be denoted by $F(T_i)$. Of the three performance measures mentioned earlier, only the first two will be considered, namely, minimizing the schedule length, $w = MAX\{F(T_i)\}$ for all tasks T_i and minimizing the mean weighted flow time, $mwft = \sum_{i=1}^{n}(W_i F(T_i))$. When the weights W_i are all the same, the mean weighted flow time will be referred to simply as mean flow time or mft.

1.2.2 Problem Classification and Reduction

It is useful to be able to classify combinatorial problems according to their degree of complexity. Since the reports of Cook (1971) and Karp (1972) classification into the classes P and NP have become widely used. These problem classes were originally defined in connection with language recognition problems but have found application in other disciplines. An algorithm is said to be polynomial-bounded if its worst-case complexity is bounded by a polynomial function of the input size, that is, if there is a polynomial g such that for each input of size n the algorithm terminates after at most g(n) steps. A problem is

polynomial-bounded if there is a polynomial-bounded algorithm for it. The reason for the criterion of polynomial-boundedness is that, in general, a polynomial-bounded algorithm will complete its task in reasonable time. For problems, polynomial-boundedness approximates one's intuitive notion of tractability while a problem whose solution requires exponential time may be looked upon as intractable. Roughly speaking, the class P may be identified with the class of problems for which polynomial-bounded algorithms exist, whereas all problems in the class NP can be solved by polynomial-depth backtrack search.

In this context, all problems must be stated in terms of recognition problems that require a yes/no answer. For scheduling problems, the transformation is accomplished by specifying a target, D, on the performance measure and asking if there is a schedule with performance measure not exceeding D.

The following concept of problem reduction is essential for further understanding of the P and NP classes. A problem, X, is <u>reducible</u> to problem Y if for any instance of X an instance of Y can be constructed in polynomial-bounded time such that solving the instance of Y will solve the instance of X as well. Problem Y is <u>NP-complete</u> if Y is in the class NP and X is reducible to Y for every problem X in NP.

A good (that is, polynomial-bounded) algorithm for any NP-complete problem could be used to construct good algorithms for every problem in NP. However, no such algorithm has been found and it seems likely that none ever will. Many problems of a combinatorial nature (Garey & Johnson, 1978a) and in particular many scheduling problems (Ullman, 1974, 1975; Garey, Johnson & Sethi, 1976; Gonzalez & Sahni, 1978; Lenstra & Rinooy-Kan, 1978; etc) are in the class NP-complete. In chapters 3, 4 and 5 several new additions are made to this ever growing list.

Generally, scheduling problems trivially belong to the class NP. (Ullman, 1974) Consequently, in order to prove NP-completeness it is sufficient only to demonstrate a polynomial reduction from a known NP-complete problem. The known NP-complete problems which are used in subsequent chapters are PARTITION and 3-PARTITION, defined (Gonzalez, Johnson & Sethi, 1976) as follows:

PARTITION:

Given a set $\{a_1,\ldots,a_n\}$ of n non-negative integers whose sum is 2K, does there exist a subset u of the indices $\{1,\ldots,n\}$ such that $\Sigma_{i\in u}(a_i) = K$?

3-PARTITION:

Given a set $\{a_1,\ldots,a_{3n}\}$ of 3n non-negative integers whose sum is nK, and $K/4 < a_i < K/2$, does there exist a partition of the integers into n disjoint groups of

three elements each such that each group sums exactly to K?

So far, questions concerning the actual nature of the computer performing the algorithms and the manner in which the size of a problem's input is to be measured have been left untouched. Experience indicates that the actual nature of the computer is relatively unimportant as far as algorithm complexity is concerned so long as the assumptions made are reasonable. It is assumed here that the hypothetical computer is capable of executing conventional instructions such as integer arithmetic, numerical comparisons and branching operations. Normally, each instruction takes one unit of time and unlimited random access memory is available. However, the actual amount of the memory required for each algorithm will be determined to within a constant factor.

The question of measuring input size is much more difficult. Some refinement in the classification, NP-complete, is possible according to the manner in which data is encoded. For scheduling problems the input size is normally taken to be either the number of tasks in the system or the sum of execution times of all tasks in the system. Usually, NP-complete results using the former measure are weaker than those using the latter. Garey and Johnson (1978) present a comprehensive discussion of this topic. Following their terminology, 3-PARTITION and problems

shown NP-complete by reduction from 3-PARTITION in this thesis are strongly NP-complete as opposed to PARTITION and problems reduced from PARTITION.

1.2.3 Solution Techniques

The techniques that have been found useful in solving scheduling and indeed combinatorial optimization problems in general include linear programming, recursion and enumeration, random sampling and heuristics. Baker (1974) and Lawler (1976) discuss these methods in some detail. Recursion and enumeration include dynamic programming (Sahni, 1976; Baker & Schrage, 1978), branch and bound (also called backtrack programming) (Ignall & Schrage, 1965; Kohler & Steiglitz, 1974) and neighbourhood search techniques (Kohler & Steiglitz, 1971). Several of these techniques are illustrated by material in ensuing chapters.

A technique that seems peculiar to scheduling is that of adjacent pairwise interchange as exemplified by Johnson's (1954) algorithm. In this case, an arbitrary schedule is modified and improved by considering the effect on the schedule's performance of interchanging two adjacent tasks.

As noted earlier, for those problems which have been proved NP-complete, it is highly unlikely that good algorithms will ever be found. Hence, emphasis shifts to the design of heuristic methods. Rather than ensure that an

optimal solution is obtained, such methods try to provide reasonably good solutions in polynomial time. An important analysis problem for such algorithms is the determination of how far, in some sense, from the optimal the heuristic can get. Well-known heuristics include keep-busy or greedy-processor scheduling, priority-list or simply list scheduling, and SPT and LPT rules (Coffman, 1974). In greedy-processor schedules, a task is assigned to a processor as soon as the processor is idle and the task is ready to be executed, that is, all preceding tasks have been executed. List schedules are keep-busy schedules in which tasks are selected for execution on available processors according to a pre-determined priority rating. SPT (LPT) schedules are list schedules in which highest priority is given to the task with the shortest (longest) execution time.

A fairly recent technique for combinatorial optimization problems is that of approximation algorithms (Sahni, 1975; Ibarra & Kim, 1975). Such algorithms are guaranteed to produce in polynomial time, solutions that are arbitrarily close to the optimal. The application of this idea to scheduling problems has not received much attention (Sahni, 1976, gives some results for independent tasks) and this seems to be a promising area for further investigation. Note, however, the results of (Garey & Johnson, 1978) which indicate that for strong NP-complete problems, fully

polynomial approximation schemes cannot be obtained.

1.3 Outline of Thesis

For most models, the scheduling problem in its most general form has been shown NP-complete, as previously noted. However, the development and analysis of heuristic methods as well as the classification of interesting and useful restricted models remains an important study. In this thesis, several scheduling models are considered and a number of restricted types of scheduling problems are solved.

Chapter 2 is concerned with the question of determining tight bounds for the lengths of list schedules of independent tasks on m identical processors in terms of the optimal schedule length. The relevant literature is surveyed and an open problem of Graham (1974) is partially solved.

Chapter 3 introduces the processor bound systems of which the flow shop and open shop considered in the following two chapters may be considered special types. A survey in this area is provided. Then it is shown that the problem of scheduling unit execution time processor bound systems is NP-complete even when the precedence constraints are restricted to chains. In addition, a dynamic programming method is suggested for scheduling such systems, and this leads to a consideration of terminal subset enumeration, an

interesting combinatorial problem in itself.

After a brief survey of pertinent results, a number of special cases of the three-stage flow shop minimal length scheduling problem is shown to be NP-complete in Chapter 4. Then polynomial algorithms are presented for two other cases. The design of the algorithms illustrates the use of adjacent pairwise interchange and enumeration.

The open shop is dealt with in Chapter 5. After a brief introduction and survey, the major part of the chapter is devoted to a reduction from 3-PARTITION which shows the two-processor minimal mean flow problem to be NP-complete. The chapter is concluded with two theorems bounding the performance of arbitrary and SPT schedules in terms of the optimal mean flow time.

The final chapter summarizes the results and discusses several suggestions for further research.

Chapter Two

BOUNDS ON SCHEDULES FOR INDEPENDENT TASKS

Perhaps, one of the most basic scheduling problems is that of minimizing the schedule length for a set of independent tasks, (that is, no precedence constraints) on parallel identical processors. Assume the system consists of n tasks, T_i, $1 \leq i \leq n$, with execution times t_i, $1 \leq i \leq n$, and m identical processors, P_j, $1 \leq j \leq m$. In the single-processor case, m = 1, the schedule length is constant for all sequences of the n tasks and there is no optimization problem, whether schedules are allowed to be preemptive or non-preemptive. However, for $m \geq 2$ processors, the problem can be solved easily for preemptive schedules while the case for non-preemptive schedules is extremely difficult. This chapter deals with the problem of providing tight bounds for the list scheduling heuristic for the non-preemptive problem.

2.1 Survey

The well-known linear algorithm for the preemptive problem was first reported by McNaughton (1959) and for quite a while no comparative results for the non-preemptive case were known. The reason for this became apparent when the non-preemptive problem was shown to be NP-complete

(Bruno, Coffman & Sethi, 1974). As for heuristics, list schedules and, in particular, LPT schedules are known (Graham, 1974; Baker, 1974) to have good performance. Sahni (1976) applied dynamic programming and developed an approximation algorithm for this problem. More recently, Coffman, Garey and Johnson (1978) obtained better performance than LPT, in general, with their algorithm which was developed with ideas from the theory of bin-packing.

Let w_o be the length of an optimal non-preemptive schedule, Z_o, and let w be the length of an arbitrary list schedule, Z. Graham (1969, 1972, 1974) showed that $w/w_o \leq 2 - 1/m$. This general bound is found to apply over a wide range of values of the parameters of the system, namely, task times, number of processors, and priority lists used. In particular, when the priority list is modified and the other parameters are kept constant this bound is achievable even if the ratio of maximum and minimum task execution times is never more than 4 (Graham, 1974). Graham left as an open problem the improvement of the bound for lower execution time ratios.

Let the ratio between the longest execution time and the shortest be $r \geq 1$. For $r = 1$, every task in the system has the same execution time and, hence, every list schedule is optimal. Thus, in this case $w/w_o = 1$. It seems reasonable, therefore, to expect that as r approaches 1, the bound on w/w_o can be reduced below $2 - 1/m$ which applies in

general. (One may think of systems with small ratio, r, as systems with similar tasks.)

In this chapter, the effects of lowering the maximum execution time ratio, r, are studied and tighter bounds on list schedule length as compared to the optimal length are derived as follows:

(1) For $r \leq 3$,

$$w/w_o \leq \begin{cases} 2 - 1/(3\lfloor m/3 \rfloor), & \text{for } m \geq 6, \\ 17/10, & \text{for } m = 5, \\ 5/3, & \text{for } m = 3, 4. \end{cases}$$

(2) For $r \leq 2$,

$$w/w_o \leq \begin{cases} 5/3 - 1/(3\lfloor m/2 \rfloor), & \text{for } m \geq 4, \\ 3/2, & \text{for } m = 2, 3. \end{cases}$$

The result for $r \leq 3$ is proved by contradiction. It is shown by induction on the number of processors that there cannot exist any pair of schedules Z and Z_o whose finish time ratio w/w_o is larger than the stated value. If there exists such a pair of schedules, which is referred to as a counter-example, for m processors, then by some simple "normalization" operations it can be modified to produce a counter-example for m - 3 processors. But, as will be seen, no counter-examples exist for small values of m. For the second case where $r \leq 2$, a similar technique is again employed. In both cases, examples which achieve the stated bounds are given.

2.2 Normalization

In this section, several lemmas, which will be useful in deriving the bound $w/w_o \leq (2 - 1/p)$ for some value of p, are given. Hence the lemmas use $(2 - 1/p)$ as a tentative bound without specifying a value for p. Since the bound $(2 - 1/m)$ is known, p must be less than or equal to m.

In the following, the longest and shortest execution times will be denoted by t_L and t_s respectively. The starting time of task T_L, the task with execution time t_L, will be denoted by y, that is, $S(T_L) = y$. If several tasks have the same execution time, t_L, one will be selected and it will be clear from the context which one it is.

By a counter-example is meant a given task system with a list schedule Z and an optimal schedule Z_o such that $w/w_o > 2 - 1/p$. The effect of applying the following lemmas, a process called normalization, is to transform any given counter-example into another counter-example with a specific structure.

Lemma 2.1: For a given set of tasks, T_i, $1 \leq i \leq n$, there exists a list schedule with longest finish time (of all list schedules) for which $F(T_L) = w$. In other words, T_L is the last task or one of the last tasks to be finished.

Proof: Let Z' be a list schedule with longest finish time w' and $F'(T_L) \neq w'$ for any task T_L with the maximum

execution time t_L. ($S'(T_i)$ and $F'(T_i)$ are the starting and finishing times of T_i in schedule Z'). Since there is no precedence relationship between the tasks, the tasks on each processor can be sorted in non-decreasing order. This operation does not change the length of Z'.

Now, let P_j be a processor which finishes last in Z'. Then, there exists T_k executed on P_j with $F'(T_k) = w'$. Also, assume that P_i is the processor that executes a task T_L with processing time t_L so that P_i is idle after time $F(T_L)$. Because of the manner in which list schedules are built there can be no idle time in Z' before $S'(T_k)$. Hence $S'(T_k) \leq F'(T_L)$ and $t_L + t_k \geq w' - S'(T_L)$. Now, form a new schedule Z with $F(T_L) = w$ and $w \geq w'$ as follows. Replace T_L on P_i by T_k giving a schedule in which the first idle time occurs at $MIN\{S'(T_k), F(T_k)\}$, where $F(T_k)$ is of course the finishing time of T_k in the new schedule. Make T_L the last task on processor P_i if $F(T_k) \leq S'(T_k)$; otherwise, make T_L the last task on processor P_j. This yields the new schedule Z with $w \geq w'$ and $F(T_L) = w$. (See Figure 1. Hatched areas indicate idle periods on processors.) Since by hypothesis Z' has longest finishing time, it follows that $w = w'$. ∎

Thus, without loss of generality, one may assume that the worst list schedule has the form depicted in Figure 2, where U indicates the total busy times on other processors during the execution of T_L. There can be no idle periods in the first $y = S(T_L)$ time units. Possible idle periods in the

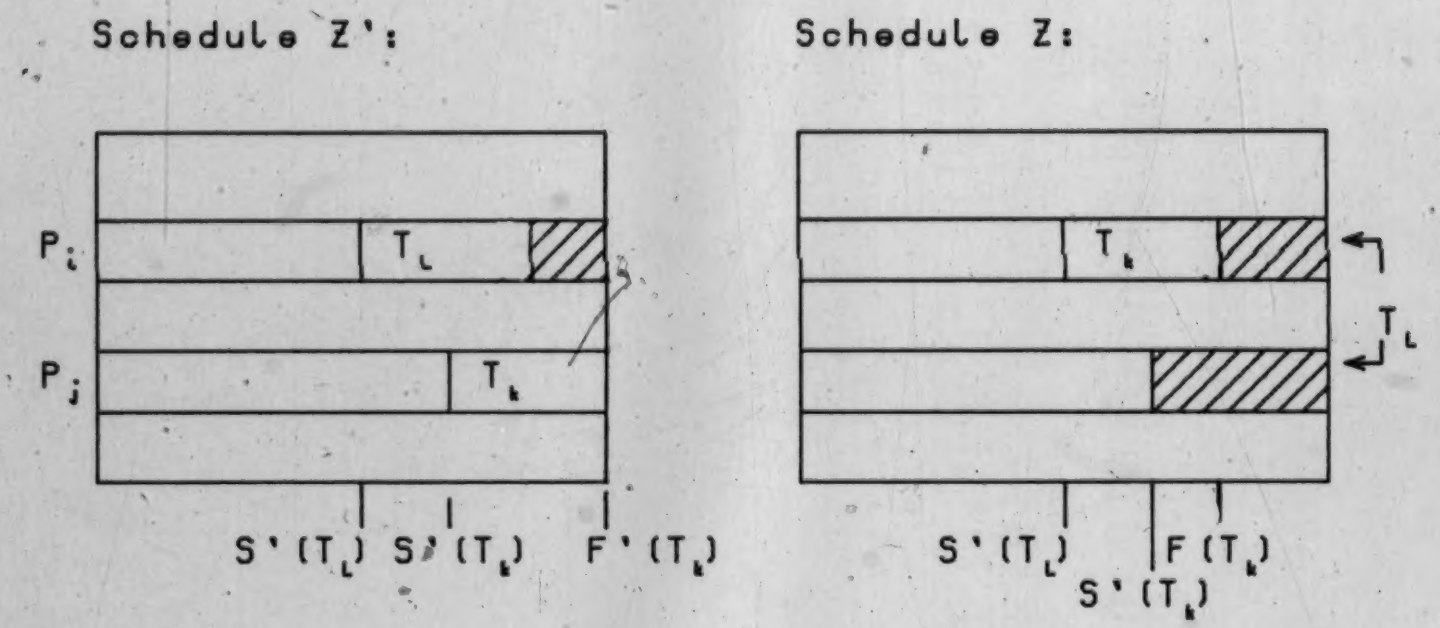


FIGURE 1: Schedules for Lemma 2.1

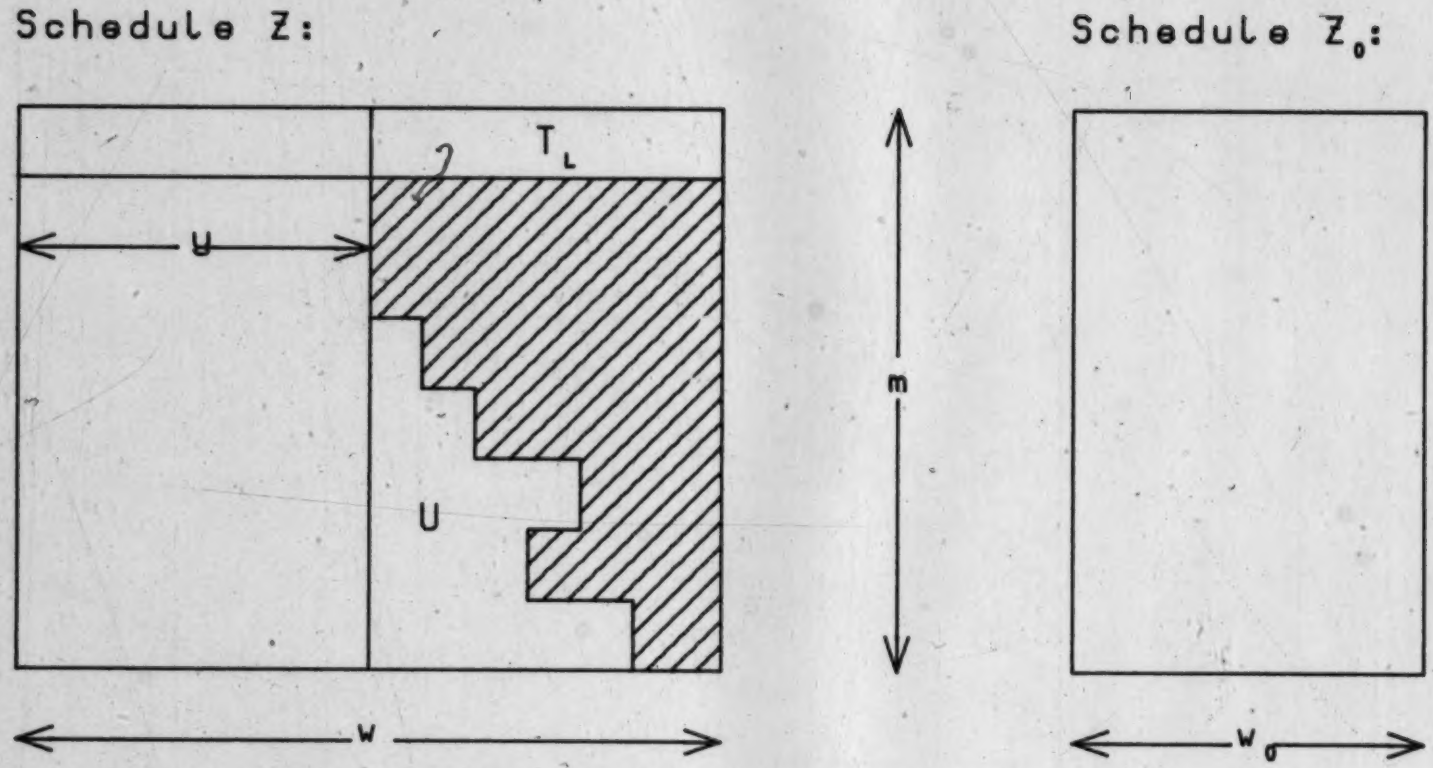


FIGURE 2: Structures of list schedule Z and optimal schedule Z_0

optimal schedule Z_o are not indicated in the Figure. For the rest of this chapter assume all counter-examples to be represented as above.

Lemma 2.2: If the worst list schedule Z with the corresponding optimal schedule Z_o satisfies $w/w_o > 2 - 1/p$ then $w_o < p(m-1)t_L/((p-1)m)$ for $m \geq 2$.

Proof: Refer to Figure 2. By considering the "processor-busy" areas in Z and Z_o,

$$mw_o \geq my + t_L + U = my + mt_L - (m-1)t_L + U$$
$$\Rightarrow mw_o \geq mw - (m-1)t_L$$
$$\Rightarrow 1 + (m-1)t_L/(mw_o) \geq w/w_o.$$

Since $w/w_o > 2 - 1/p$, $1 + (m-1)t_L/(mw_o) > 2 - 1/p$ from which the result follows. ∎

Corollary: If the conditions of Lemma 2.2 hold and $p = 3\lfloor m/3 \rfloor$, then $w_o < 6t_L/5$ for $m \geq 3$ and $w_o < 21t_L/20$ for $m \geq 6$.

Proof: There are three cases depending on the value of $m \pmod 3$.

(1) $m = 3i$, $p = 3i$. From Lemma 2.2, $w_o < t_L$.

(2) $m = 3i+1$, $p = 3i$. From Lemma 2.2,
$w_o < (1 + 1/(m^2-2m))t_L < 9t_L/8$ for $m \geq 4$.

(3) $m = 3i+2$, $p = 3i$. From Lemma 2.2,
$w_o < (1 + 2/(m^2-3m))t_L < 6t_L/5$ for $m \geq 5$.

Similarly for $m \geq 6$. ∎

Lemma 2.3: Given a counter-example with list schedule Z' and optimal schedule Z'_o we can obtain another counter-example Z, Z_o for which $w = 2w_o - 1$ and $y = w_o - 1 + g$, where $t_L = w_o - g$, $0 \leq g \leq 1$.

Proof: This requires only the application of a scaling operation to all task times. From the definition of counter-examples for Z', $2 - 1/p < w'/w'_o \leq 2 - 1/m$. Let $w'/w'_o = 2 - 1/d$ for some real number d, $p < d \leq m$. Multiplying all t_i by d/w'_o and using the same schemes as in Z' and Z'_o gives Z, Z_o with $w_o = d$, $w/w_o = 2 - 1/w_o$, $w = 2w_o - 1$ and the ratio $w/w_o = w'/w'_o > 2 - 1/p$. Furthermore, since (see Figure 2) $w = y + t_L$ and $t_L \leq w_o$, $y = w_o - 1 + g$ and $t_L = w_o - g$ for some g, $0 \leq g \leq 1$. ∎

In the following it is assumed that the scaling operation of Lemma 2.3 has beeen applied to the schedules.

Lemma 2.4: Let Z and Z_o be a counter-example for $m \geq 3$ and $r \leq 3$. Then, no processor executes more than three tasks in Z_o. Furthermore, the processor which executes T_L in Z_o does no other tasks.

Proof: From the Corollary of Lemma 2.2, $w_o < 6t_L/5$. Suppose some processor executes four tasks, $T_{i(1)}$, $T_{i(2)}$, $T_{i(3)}$, $T_{i(4)}$, in Z_o, then

$$\Sigma_{j=1}^{4}(t_{i(j)}) \geq 4t_L/3 > 6t_L/5 > w_o,$$

which is a contradiction. Also, since $w_o - t_L < t_L/5 < t_L/3 \leq t_i$, $1 \leq i \leq n$, the processor which

executes T_L in Z_o can process no other task. ∎

Lemma 2.5: If list schedule Z and optimal schedule Z_o form a counter-example on m processors, $m \geq 7$, and $p = 3\lfloor m/3 \rfloor$, then, $t_L \geq y$.

Proof: Since $y = 2w_o - 1 - t_L$ (by Lemmas 2.1 and 2.3; see Figure 2) $t_L \geq y$ if and only if $t_L - (2w_o - 1 - t_L) \geq 0$ or $1 + 2(t_L - w_o) \geq 0$. The last inequality is easily shown to be true as follows. By Lemma 2.2,

$$w_o < p(m-1)t_L/((p-1)m)$$

$$\Rightarrow 1 + 2(t_L - w_o) > 1 + 2[mw_o(p-1)/(p(m-1)) - w_o]$$

$$= 1 - 2w_o(m-p)/(p(m-1))$$

$$\geq 1 - 2m(m-p)/(p(m-1)),$$

(since $w_o \leq m$ after scaling by Lemma 3.3). Thus, it is sufficient to show that $G = 2(m-p)m/(p(m-1)) \leq 1$.

For $p = m$, $G = 0$.

For $p = m - 1$, $G \leq 1$ as $2m \leq m^2 - 2m + 1$ for $m \geq 6$.

For $p = m - 2$, $G \leq 1$ as $m^2 - 7m + 2 \geq 0$ for $m \geq 7$.

Hence, for $m \geq 7$, $1 + 2(t_L - w_o) \geq 0$ and $t_L \geq y$. ∎

A task will be referred to as k-partnered in Z_o if it is executed with k other tasks on the same processor in Z_o. For a list schedule Z the definition of k-partners will be the same except that T_L is not counted as a partner of any other task.

Lemma 2.6: Let Z and Z_o represent a counter-example as before, with $m \geq 7$, $r \leq 3$, and $p = 3\lfloor m/3 \rfloor$. Then, there

exists a counter-example consisting of schedules Z' and Z'_o with the same finish time ratio $w'/w'_o = w/w_o$ for which the following properties hold:

(i) In both Z' and Z'_o any 2-partnered task is smaller (in execution time) than any 1-partnered task which in turn is smaller than any 0-partnered task.

(ii) In Z'_o, if T_i and T_j are 1-partnered and are executed on the same processor, then if $t_i \geq t_j$ then $t_i \geq w_o/2$.

Proof: If the properties do not hold already, a number of operations are performed on the schedules and task times while keeping the finish time ratio unchanged.

(i) First consider the list schedule Z. With reference to Figure 2 any modification on the tasks except T_L in Z which does not include idle time before time $y = 2w_o - 1 - t_L$ will yield a list schedule with length not less than w.

Since any three tasks have total execution time at least as big as t_L and by Lemma 2.5 $t_L \geq y$, it can be ensured that the smallest tasks are the 2-partnered tasks by exchanging any larger 2-partnered tasks with smaller 1- or 0-partnered tasks.

Furthermore, any 0-partnered task T_k in Z has the property that $t_k \geq y$. Hence if there is a larger 1-partnered task T_j, where $t_j > t_k \geq y$, then exchanging T_k and T_j will still give a schedule with finish time w. Thus, the tasks in Z can be rearranged accordingly (yielding schedule Z') such

that a 0-partnered task is no smaller than a 1-partnered task which in turn is no smaller than any 2-partnered task. This concludes the proof of part (i) for Z'.

For Z_o it is first shown that a 0-partnered task in Z' is also a 0-partnered task in Z_o and vice versa. Let T_k be a 0-partnered task in Z', then $t_L \geq w_o - 1$, and $w_o - t_k \leq 1 < t_L/3$. (From Lemmas 2.3 and 2.5, after the scaling operation, $t_L \geq y \geq w_o - 1 > p - 1 \geq m - 3$; hence $t_L > 3$ for $m \geq 6$.) Hence no other task can be executed on the same processor with T_k in Z_o. That is, a 0-partnered task in Z is a 0-partnered task in Z_o. Now suppose there is a 0-partnered task, T_k, other than T_L in Z_o which is not 0-partnered in Z'. As in the proof of Lemma 2.2 a contradiction is obtained by considering the processor-busy areas of Z_o and Z' (see Figure 2). Suppose there are $u \geq 0$ 0-partnered tasks common to Z' and Z_o. Let $q = m - u - 2$ be the number of processors (excluding the processors which execute T_k and T_L) that execute 2- and 1-partnered tasks in Z_o. The total execution time on the q processors in Z_o is at most $w_o q$. Since T_k has a partner in Z', its partner must take at least $t_L/3$. Therefore, out of the total execution time on the q processors at most $w_o q - t_L/3$ must be shared between $q + 1$ processors in Z'. Hence,

$$qw_o - t_L/3 > (q + 1)(w_o - 1)$$

$$\Rightarrow q + 1 > w_o + t_L/3$$

$$\Rightarrow (m - 2) + 1 \geq q + 1 > w_o + t_L/3 \geq w_o + 5w_o/18$$

$\Rightarrow m - 1 > 23w_o/18 \geq 23(m - 2)/18$ (since $w_o > p \geq m - 2$)

$\Rightarrow 18(m - 1) > 23(m - 2)$ or $5m < 28$,

giving a contradiction for $m \geq 6$. Hence, a 0-partnered task in Z_o must also be a 0-partnered task in Z' (except for T_L). It follows that the 0-partnered tasks are the largest in Z_o, since they are the largest in Z'.

The construction of schedule Z_o' can now be specified. Let T_a be the smallest 1-partnered task in Z_o, T_b its (larger) partner and T_c the largest 2-partnered task in Z_o. If $t_a \geq t_c$, then (i) holds for Z_o and Z_o' is identical to Z_o. Suppose $t_a < t_c$. Note that $t_c \leq w_o - 2t_L/3$ since $t_L/3$ is the minimum possible length for each of its partners. Hence, $t_a < w_o - 2t_L/3$. Now, increase t_a to $w_o - 2t_L/3$ exactly thus forcing $t_a \geq t_c$. Since $t_a + t_b \leq w_o$, t_b must be reduced, if necessary, to $2t_L/3$. This operation of setting $t_a = w_o - 2t_L/3$ and $t_b = 2t_L/3$ does not change w_o. In Z', increasing any task time will not reduce w and thus w/w_o does not decrease. Also, it is obvious that T_b can be a 2- or 1-partnered task in Z' and decreasing a 2-partnered task's time to $2t_L/3$ will not reduce w. For a 1-partnered task, even if the other partner has minimum time, $t_L/3$, their total length must be at least $t_L \geq y$, ensuring no reduction in w. The operation can be repeated until there is no 1-partnered task that is smaller than a 2-partnered task in Z_o, thus obtaining schedule Z_o'. This concludes part (i) for Z_o'.

(ii) Here, note that if the larger of a pair of tasks executed on the same processor in Z_o is less than $w_o/2$ then there must be idle time on that processor. Increase the larger task's time by the amount of idle time on the processor. Again, w_o does not increase and w does not decrease. ∎

The last of the normalization results imposes an order on the 1-partnered tasks in both schedules Z and Z_o of a counter-example.

Lemma 2.7: Let Z and Z_o be a counter-example as before. The 1-partnered tasks in Z and Z_o can be ordered so that the largest is a partner of the smallest, the next largest a partner of the next smallest and so on. The result remains a counter-example.

Proof: The schedules are modified with the aid of two operations similar to those employed by Graham (1974).

Operation I:

Sort the tasks on each processor in non-decreasing order. This does not change the schedule's length.

Operation II:

Let T_i and T_j be partners, and similarly for T_g and T_h. (See Figure 3.) If $t_i < t_g$ and $t_j < t_h$, exchange T_j and T_h.

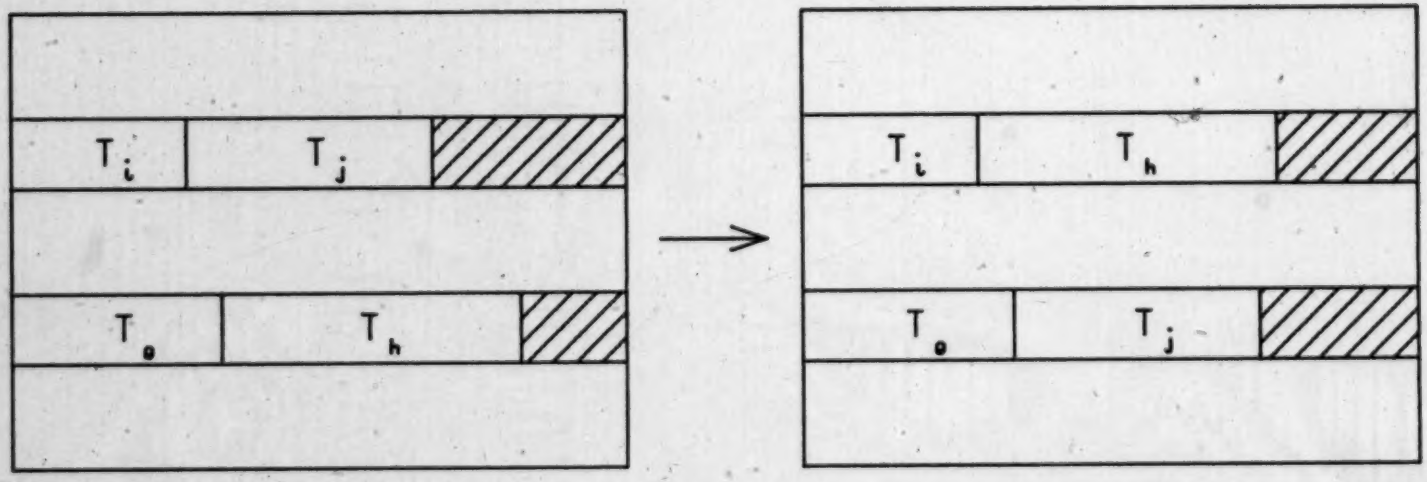


FIGURE 3: Operation II of Lemma 2.7

Apply both operations iteratively until there are no indices, g, h, i, j, for which operation II can be applied. It is easy to see that the effect is to delay the first occurrence of idle time and possibly decrease the schedule's length. Also, only a finite number of operations will be done (Graham, 1974). Since Z_o is optimal, w_o will remain unchanged. Note that in Z, T_L is not considered for above operations (recall that T_L is not considered a partner of any other task). Hence, the relevant effect of operations I and II on Z is the possible delay of first occurrence of idle time y. Since y is not decreased, w will not be decreased.

This concludes the presentation of the normalization lemmas. Let $p = 3\lfloor m/3 \rfloor$, $r \leq 3$, $m \geq 7$. Then, if counter-examples exist, consider the worst case. By application of the previous lemmas, it can be transformed into a counter-example with schedules Z and Z_o for which $w = 2w_o - 1$, the 2-partnered tasks are the smallest in both Z and Z_o, and the 0-partnered are the largest. Furthermore the 1-partnered tasks in both schedules are paired in an orderly manner and can be represented as in Figure 4. Note that the set of tasks $\{T_i \mid 1 \leq i \leq n\} - \{T_L\}$ occupy one more processor in Z than in Z_o. Also, recall (from proof of Lemma 2.6) that a task is 0-partnered in Z_o if and only if it is 0-partnered in Z. Hence, this 1-processor gain must be achieved by having a difference of exactly 6 tasks in the

division between 1- and 2-partnered tasks. It follows that there are at least two processors in Z_o that perform three tasks each (exactly two more than in Z, not counting T_L in Z).

Furthermore, it is guaranteed that at least three processors in Z_o perform two tasks each for the following reason. Suppose there are $k < 3$ processors that perform two tasks each in Z_o. Then, the number of 1-partnered tasks is $2k$ in Z_o and $2k + 6$ in Z, where the additional 6 tasks in Z are 2-partnered in Z_o. Now, for $k < 3$, these six tasks are greater in number than $k + 3$, the number of processors which execute the $2k + 6$ tasks in Z. Hence, two of the six tasks which are 2-partnered in Z_o must be executed on the same processor in Z. Their total time is at most $2t_c = 2(w_o - 2t_L/3)$, using t_c of Lemma 2.6. Using $w_o > p$ (by Lemma 2.3) and the second part of the corollary to Lemma 2.2, it is easily shown that this number is less than $w_o - 1$, and hence is not sufficient for y (that is, contradicting the fact that there is no idle time in Z before time y).

In the next section, tight bounds for systems with $r \leq 3$ and $r \leq 2$ are proved using normalized counter-examples.

2.3 Bounds for Similar Tasks

The following relations which apply to a normalized counter-example should be kept in mind while studying subsequent proofs in this section.

(1) $w_o < p(m - 1)t_L/((p - 1)m)$ for $m \geq 2$. (Lemma 2.2)

(2) $w_o < 6t_L/5$ for $p = 3\lfloor m/3 \rfloor$, $m \geq 3$. (Corollary of Lemma 2.2)

(3) $w = 2w_o - 1$, $y = w_o - 1 + g$, $t_L = w_o - g$, where $0 \leq g \leq 1$. (Lemma 2.3)

(4) $p < w_o \leq m$. Hence, $w_o \geq 6$ for $p = 3\lfloor m/3 \rfloor$, $m \geq 7$. (follows from Lemma 2.3, since for a counter-example, $2 - 1/p < w/w_o \leq 2 - 1/m$).

(5) $w_o \geq t_L \geq y \geq w_o - 1$ for $p = 3\lfloor m/3 \rfloor$, $m \geq 7$. (from Lemmas 2.3 and 2.5).

2.3.1 Tasks With Largest Execution Time Ratio ≤ 3

Informally, the proof for the bound for $m \geq 6$ runs as follows: If a counter-example exists for any m, $m \geq 6$, then another can be constructed for $m - 3$ processors (Lemma 2.8). Hence, by running through a series of constructions a counter-example for $m = 6$, 7 or 8 can be produced. But in Lemma 2.9 it is shown that no counter-examples exist for $m = 6$, 7 or 8 thus giving a contradiction. The bound is shown to be tight by demonstrating some examples.

Lemma 2.8: Let Z, Z_o constitute a normalized

counter-example for m processors, where $p = 3\lfloor m/3 \rfloor$, $m \geq 9$ and $r \leq 3$. Then, one can construct a counter-example for m - 3 processors.

Proof: Note that all results of previous lemmas, where dependent on m, are valid for $m \geq 7$, and also there is no need to consider those cases where m is a multiple of 3 since the bound in these cases is exactly 2 - 1/m which has been proved (Graham, 1974).

The counter-example for m-3 processors is constructed in two steps. First, six of the 1-partnered tasks in Z_o are deleted (recall that the existence of at least six of them has been determined). Second, the execution times of some of the tasks are reduced slightly. It must be shown that in the resulting task system the ratio r is still less than or equal to 3, and that the resulting schedules do form a counter-example for m - 3 processors.

The tasks to be deleted are precisely those in the middle range of the 1-partnered set in Z_o, namely, $\{T_{i+u+1},\ldots,T_{i+u+6}\}$ (See Figures 4, 5). The task times of the remaining tasks will then be reduced thus:

$$t'_j = \begin{cases} t_j - 1, & \text{if } j \leq i+u, \\ t_j - 2, & \text{if } i+u+7 \leq j \leq i+2u, \\ t_j - 3, & \text{if } j > i+2u. \end{cases}$$

Note that after task time reduction, for every pair (of 1-partners) above which has not been disrupted by task

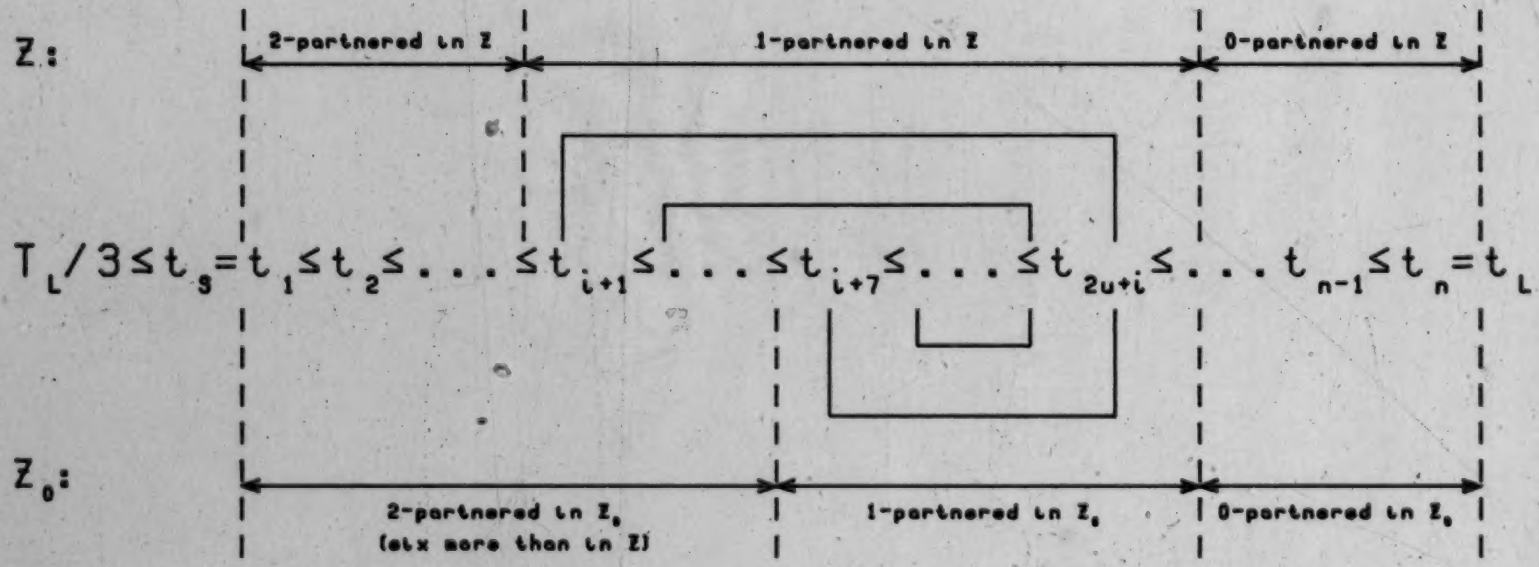

Z:
2-partnered in Z
1-partnered in Z
0-partnered in Z
$T_L/3 \le t_s = t_1 \le t_2 \le \ldots \le t_{i+1} \le \ldots \le t_{i+7} \le \ldots \le t_{2u+i} \le \ldots t_{n-1} \le t_n = t_L$
Z_0:
2-partnered in Z_0
(at x more than in Z)
1-partnered in Z_0
0-partnered in Z_0

FIGURE 4: Order of tasks in normalized counter-example, $r \le 3$

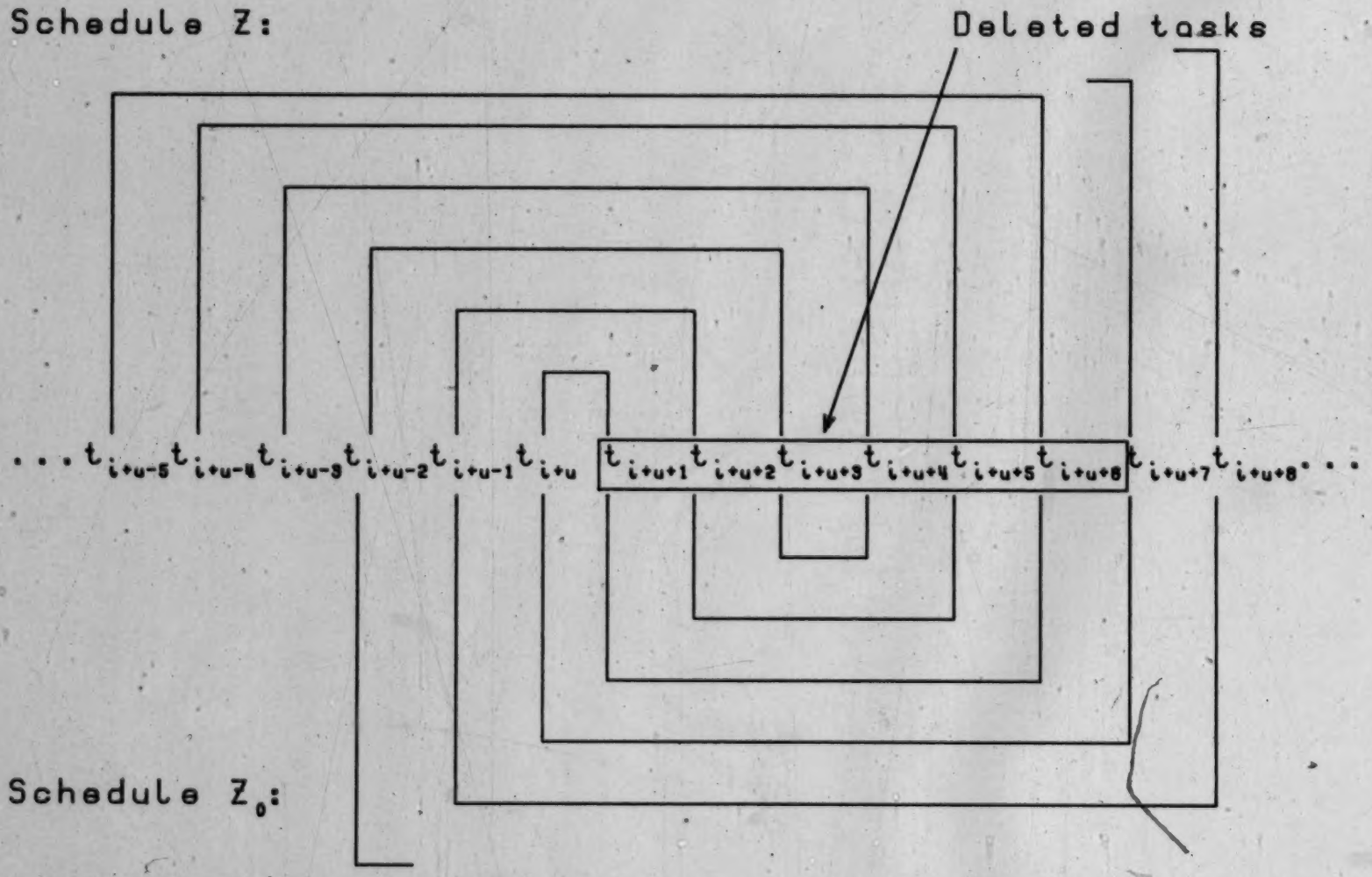


FIGURE 5: Detail at deletion area of Figure 4

deletion the total processor time is reduced by 3. But the tasks $\{T_{i+u-5},\ldots,T_{i+u}\}$ lose their partners in Z.

Now, $t_{i+u-5} + t_{i+u+6} \geq y \geq w_o - 1.$

Similarly, $t_{i+u} + t_{i+u+1} \geq w_o - 1.$

Hence, $t_{i+u-5} + t_{i+u} + t_{i+u+1} + t_{i+u+6} \geq 2w_o - 2.$

But $t_{i+u+1} + t_{i+u+6} \leq w_o$ (being a pair in Z_o).

Hence, $t_{i+u-5} + t_{i+u} \geq w_o - 2.$

Similarly, it can be shown that

$$t_{i+u-4} + t_{i+u-1} \geq w_o - 2,$$

and $$t_{i+u-3} + t_{i+u-2} \geq w_o - 2.$$

Since the above six tasks are among the tasks reduced by 1,

$$t'_{i+u-5} + t'_{i+u} \geq w_o - 4,$$

$$t'_{i+u-4} + t'_{i+u-1} \geq w_o - 4,$$

and $$t'_{i+u-3} + t'_{i+u-2} \geq w_o - 4.$$

But this is equivalent to the reduction in execution time on the other processors in Z. Hence, the tasks which are left with no partners in Z can be paired against each other. This gives two new schedules Z', Z'_o for m - 3 processors with $w' = w - 6$, $w'_o = w_o - 3$ and

$w'/w'_o = (w - 6)/(w_o - 3) = 2 - 1/(w_o - 3) > 2 - 1/(p - 3)$.

It is obvious that Z' is indeed a list schedule for the reduced set of tasks. Hence, if it can be shown that the task time ratios are less than or equal to 3 for Z', Z'_o, then they constitute a counter-example for m-3 processors. This is done by showing that (I) $t'_L = MAX\{t'_i\}$, (II) $t'_1 = MIN\{t'_i\}$, and $t'_L/t'_1 \geq 3$. Note that the $\{t'_i\}$ are in three non-decreasing sequences:

$\{t'_1, \ldots\ldots, t'_{i+u}\}$ - obtained by subtracting 1,

$\{t'_{i+u+7}, \ldots, t'_{i+2u}\}$ - obtained by subtracting 2,

and $\{t'_{i+2u+1}, \ldots, t'_n = t'_L\}$ - obtained by subtracting 3.

(I) Show $t'_L = \text{MAX}\{t'_i\}$, $1 \leq i \leq n$.

t'_L is obviously the largest element of the third sequence. Now $t_{i+6} \geq w_o/3$ (see Figure 4). t_{i+6} is the largest of the 2-partnered tasks in Z_o which are 1-partnered in Z. If it is less than $w_o/3$, then by reasoning along the lines of Lemma 2.6, part (ii), it can always be forced to equal this number at the end of normalization without changing the counter-example status of the schedules.

Hence, $t_{i+7} \geq w_o/3$, since $t_{i+7} \geq t_{i+6}$.

But $t_{i+7} + t_{i+2u} \leq w_o$ (a pair in Z_o)

Therefore, $t_{i+2u} \leq 2w_o/3$

or $t'_{i+2u} \leq 2w_o/3 - 2$

$\leq (2/3)(6t_L/5 - 2)$

(from the Corollary of Lemma 2.2)

$= (4t_L - 10)/5$.

Since $t'_L = t_L - 3 = (5t_L - 15)/5$,

$t'_L \geq t'_{i+2u}$

if and only if $5t_L - 15 \geq 4t_L - 10$ or $t_L \geq 5$.

But for $m \geq 6$ and hence for $p \geq 6$,

$6t_L/5 > w_o > p \Rightarrow t_L > 5$.

Therefore, t'_L is at least as large as t'_{i+2u}.

Furthermore, (see Figure 5)

$t_{i+u} + t_{i+u+6} \leq t_{i+u} + t_{i+u+7} \leq w_o$

and $t_{i+u} \leq t_{i+u+6}$.

Hence, $t_{i+u} \leq w_o/2$

and $t'_{i+u} \leq w_o/2 - 1 \leq (3t_L - 5)/5$.

It follows that t'_L is larger than t'_{i+u} for $t_L \geq 5$ or $m \geq 6$. Thus, t'_L is the largest of the new task times.

(II) Show $t'_1 = MIN\{t'_i\}$, $1 \leq i \leq n$.

t'_1 is obviously the smallest of the elements of the first sequence. Now, for $j > i + 2u$, task T_j is $\emptyset$-partnered in Z. Hence, $t_j \geq w_o - 1$. This implies that

$$t'_j \geq w_o - 4 \leq w_o/3 - 1 \quad (\text{for } w_o \geq 6)$$
$$\geq t_1 - 1 = t'_1.$$

Thus t'_1 is smaller than the smallest element in the third sequence. Now, t_{i+u+7} (see Figure 5) is the larger of two tasks executed on the same processor in Z_o and by Lemma 2.6, part (ii),

$$t_{i+u+7} \geq w_o/2.$$

This implies $t'_{i+u+7} \geq w_o/2 - 2$

$$\geq w_o/3 - 1 \quad (\text{true for } w_o \geq 6)$$
$$\geq t_1 - 1 = t'_1.$$

Thus t'_1 is the minimum of the new task times.

(III) $t_L/t_1 \leq 3$

$\Leftrightarrow t_L \leq 3t_1$

$\Leftrightarrow t_L - 3 \leq 3(t_1 - 1)$

$\Leftrightarrow t'_L \leq 3t'_1$.

Hence, $t'_L/t'_1 \leq 3$. ∎

Lemma 2.9: No counter-examples exist for $p = 6$, $r \leq 3$ and $m = 6$, 7 or 8.

Proof: The approach is to show that the value attainable for y in any example is not large enough to yield a counter-example.

m=6; The bound is already proved for multiples of 3 (Graham, 1974).

m=7,8; Again, note that all the normalization Lemmas hold for $m \geq 7$. After application of these lemmas a counter-example with 2u 1-partnered tasks in Z, $u \leq m$, is obtained. It is easily verified (using the relationships in Figure 4; the result is due to the fact that $u \leq m$ is too small) that at least one of the tasks $T_{i+1},\ldots,T_{i+6}$ has a partner in Z, T_a say, where T_a is either

(i) 1-partnered in Z_o and is smaller than its partner in Z_o and hence $t_a \leq w_o/2$ or

(ii) T_a is one of the above mentioned six tasks. Each of the execution times $t_{i+1},\ldots,t_{i+6}$ is less than or equal to $w_o - 2t_L/3$ (maximum for a 2-partnered task). Hence for this case t_a is also less than or equal to $w_o/2$.

Hence, $$y_o \leq (w_o - 2t_L/3) + w_o/2$$
$$= (9w_o - 4t_L)/6,$$

and $$w/w_o = (y + t_L)/w_o$$
$$\leq (9w_o + 2t_L)/(6w_o) \leq 2 - 1/6.$$

Thus, the supposed counter-example cannot be a

counter-example; contradiction. ∎

Theorem 2.1: If $r \leq 3$ and $m \geq 6$, then $w/w_o \leq 2 - 1/(3\lfloor m/3 \rfloor)$. This bound can be achieved.

Proof: By Lemma 2.8 if a counter-example exists for $m \geq 9$, then counter-examples for $m - 3$, $m - 6$, ..., (8 or 7 or 6) can be constructed. But by Lemma 2.9 no counter-examples exist for $m = 6$, 7 or 8. Contradiction.

To complete the proof of the theorem it remains to show that the bound is achievable. This is shown schematically in Figure 6. ∎

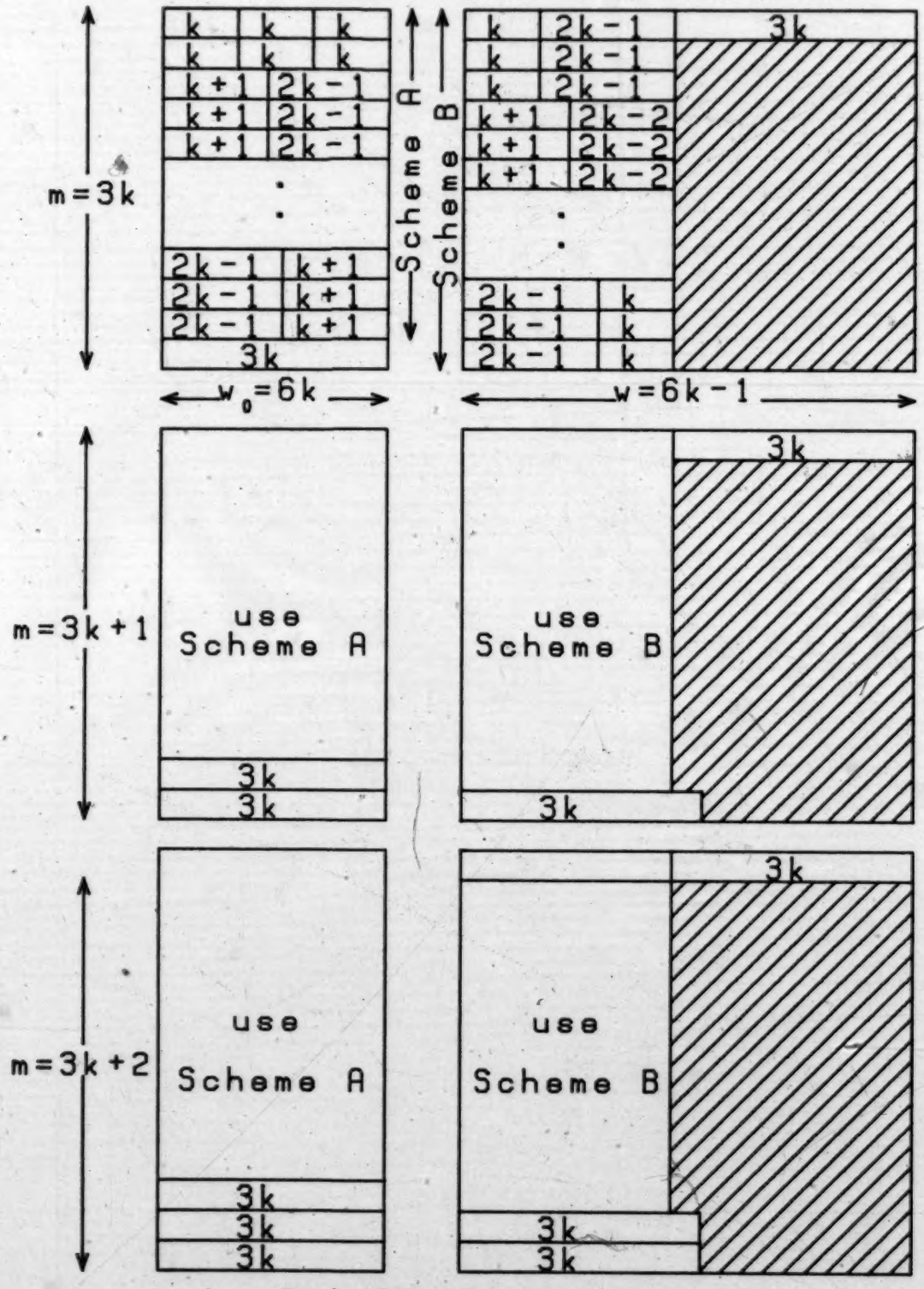
The second theorem considers the special cases $m = 4$, 5.

Theorem 2.2: Assume $r \leq 3$. If $m = 4$ then $w/w_o \leq 5/3$ (i.e. $2 - 1/3$) and if $m = 5$ then $w/w_o \leq 17/10$ (i.e. $2 - 3/10$). The bounds are tight.

Proof: Consider first the case for $m = 4$.

$m = 4$: Note that Lemmas 2.1, 2.2, 2.3 & 2.4 apply for $m = 4$. The set of tasks except T_L must be executed on one more processor in Z than in Z_o with no idle time before y. If this gain in processors is achieved by having a non-Ø-partnered task T_i in Z_o become Ø-partnered in Z, then $t_i \leq w_o - t_L/3$.

Hence, $$y \leq t_i \leq w_o - t_L/3$$



$w/w_0 = 2 - 1/3\lfloor m/3 \rfloor$, $m \geq 6$, $r = 3$

FIGURE 6: Examples which achieve the bounds of theorem 2.1

and $$w/w_o = (y + t_L)/w_o$$
$$\leq (3w_o + 2t_L)/(3w_o) \leq 2 - 1/3.$$

Hence, for a counter-example, a task which is Ø-partnered in Z is Ø-partnered in Z_o. The reverse can also be shown by argument similar to that in the proof of Lemma 2.6. As before there must be at least two processors in Z_o (exactly two more than in Z) doing three tasks. There are thus only three possibilities. See Figure 7 for sketch. Label task times V_i for tasks with two partners, W_i for the 1-partnered and X_i for the Ø-partnered.

(a) Refer to Figure 7. The six tasks V_i, $1 \leq i \leq 6$ must be executed on three processors in Z. Since there can be no idle time before y,

$$y \leq (1/3)\Sigma_{i=1}^{6} V_i \leq 2w_o/3$$

and $$w/w_o = (y + t_L)/w_o \leq (2w_o/3 + t_L)/w_o$$
$$\leq (2w_o/3 + w_o)/w_o = 2 - 1/3.$$

Hence, such a counter-example does not exist.

(b) In this case, since W_1, W_2 must be paired with at least two of the V_i the remaining tasks which are executed on the second processor give

$$y \leq (V_{i(1)} + V_{i(2)} + V_{i(3)} + V_{i(4)})/2$$
$$\leq (2w_o - 2t_L/3)/2$$

since $t_L/3$ is the minimum task time for each of the V_i already paired up. Hence, $y \leq w_o - t_L/3$ and as before $w/w_o \leq 2-1/3$, and no such counter-example can exist.

(c) Here the nine 2-partnered tasks have to be executed on four processors in Z. At least one processor must

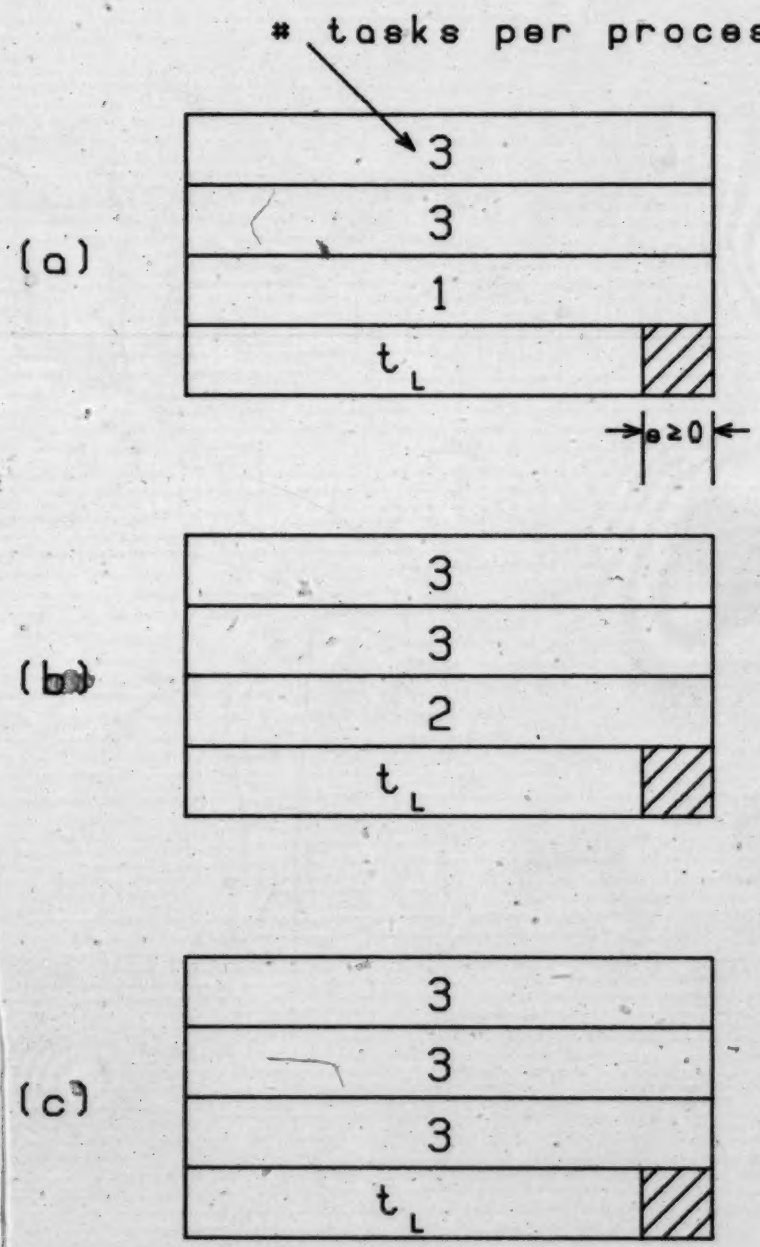


FIGURE 7:
Possible Z_0 schedules, $m=4$, $r \leq 3$

execute three of the V_i. The minimum total time for these three is t_L. Therefore, the remaining six tasks which are executed on three processors yield

$$y \leq (3(t_L + e) - t_L)/3$$

and hence $w/w_o \leq 2 - 1/3$.

Figure 8 shows the bound for m = 4 to be tight.

m = 5: The discussion at the start of case m = 4 applies leaving six possibilities for a counter-example as in Figure 9.

(a) Similar to case (a) for m = 4.

(b) Similar to case (b) for m = 4.

(c) Here there are tasks W_i, $1 \leq i \leq 4$, which are 1-partnered and V_i, $1 \leq i \leq 6$, 2-partnered in Z_o. Since all V_i, W_i must have partners (see discussion at start of m = 4) at least two of the V_i must again be partners in Z. Since any two V_i take at most $2(w_o - 2t_L/3)$,

$$w/w_o \leq (2(w_o - 2t_L/3) + t_L)/w_o$$
$$= (5w_o + e)/(3w_o).$$

As $4w_o \geq 5y$,

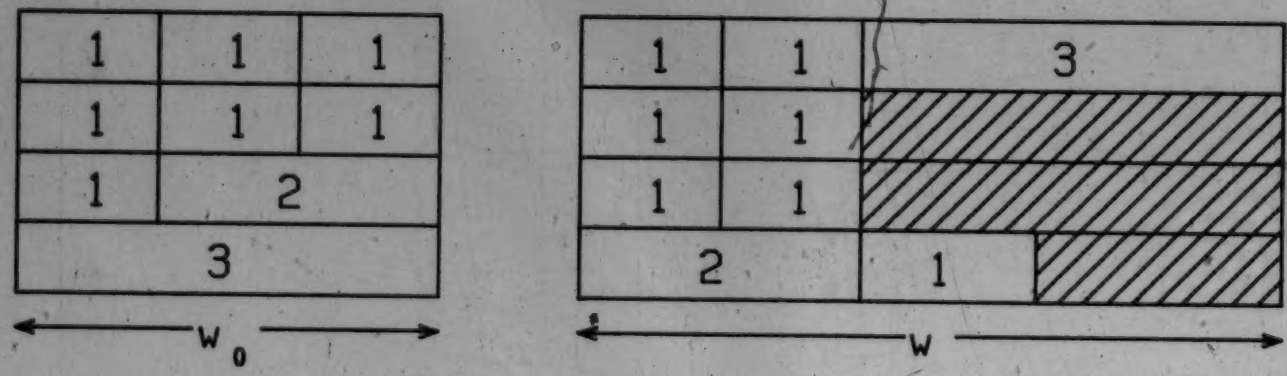
$$w/w_o = (y + t_L)/w_o$$
$$< (4w_o/5 + t_L)/w_o < 4/5 + t_L/w_o.$$

Assume $w/w_o > 17/10$ so that $17/10 < 4/5 + t_L/w_o$.

Then, $9/10 < t_L/w_o$, $e = w_o - t_L < w_o/10$,

and $w/w_o \leq (5w_o + e)/(3w_o)$
$< 5/3 + 1/30 = 17/10$. Contradiction.

Hence, $w/w_o \leq 17/10$.



$$\frac{W}{W_0} = \frac{5}{3}$$

FIGURE 8: Worst case for m=4, r≤3

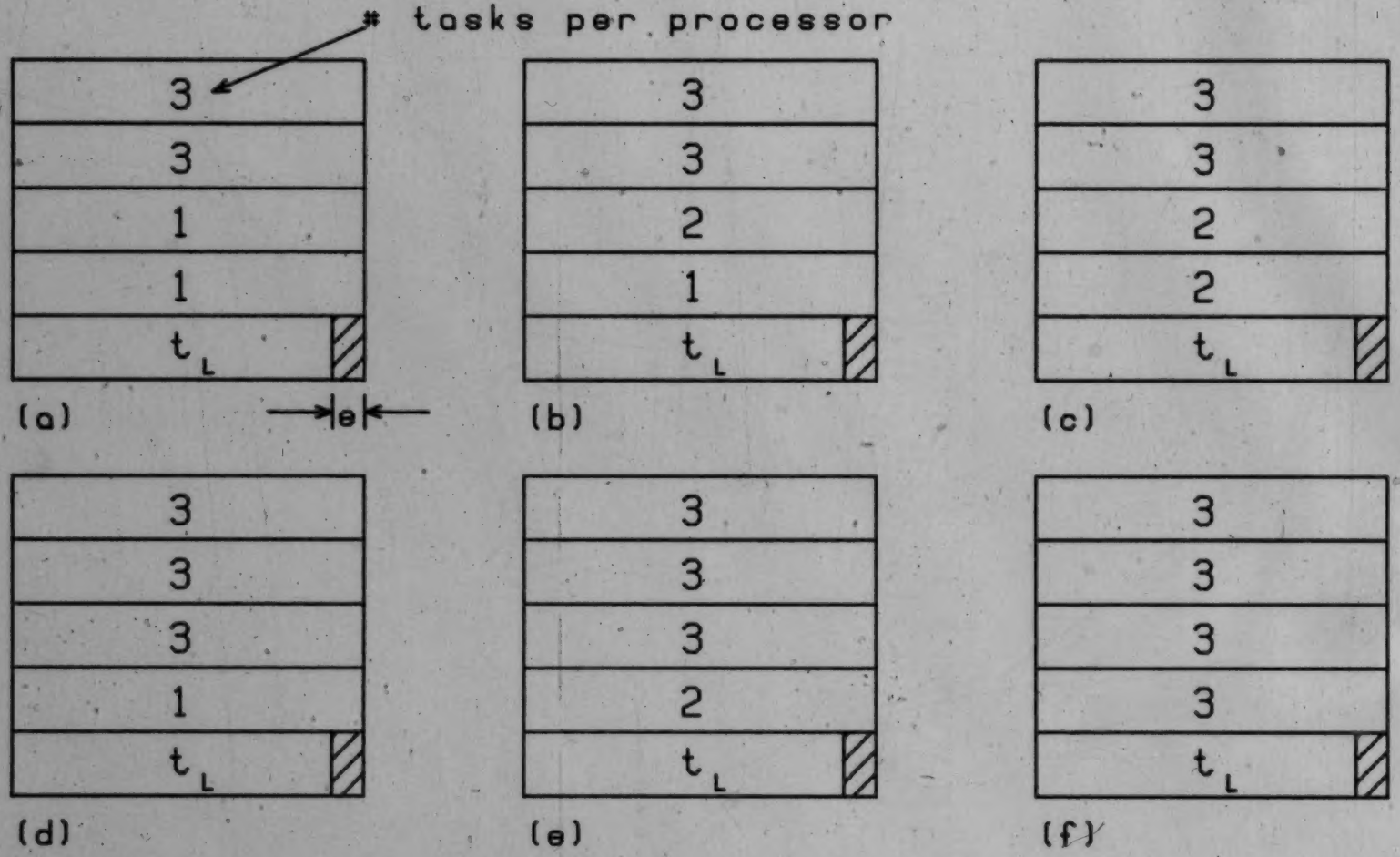


FIGURE 9: Possible Z_0 schedules, $m=5$, $r \leq 3$

(d) Similar to case (c) for $m = 4$.

(e) Here, one can see that two of the smaller tasks V_i must be paired with W_1, W_2 leaving seven tasks to be shared between three processors. Since one of these three processors must have three tasks there remains four small tasks (V_i) for two processors. Even if all previously assigned tasks have minimum execution time, $t_L/3$, this still yields

$$y \leq [3(t_L + e) - 5t_L/3]/2$$
$$= (4t_L + 9e)/6.$$
$$w/w_o \leq (10t_L + 9e)/(6t_L + 6e)$$
$$\leq 2 - 1/3 < 2 - 3/10.$$

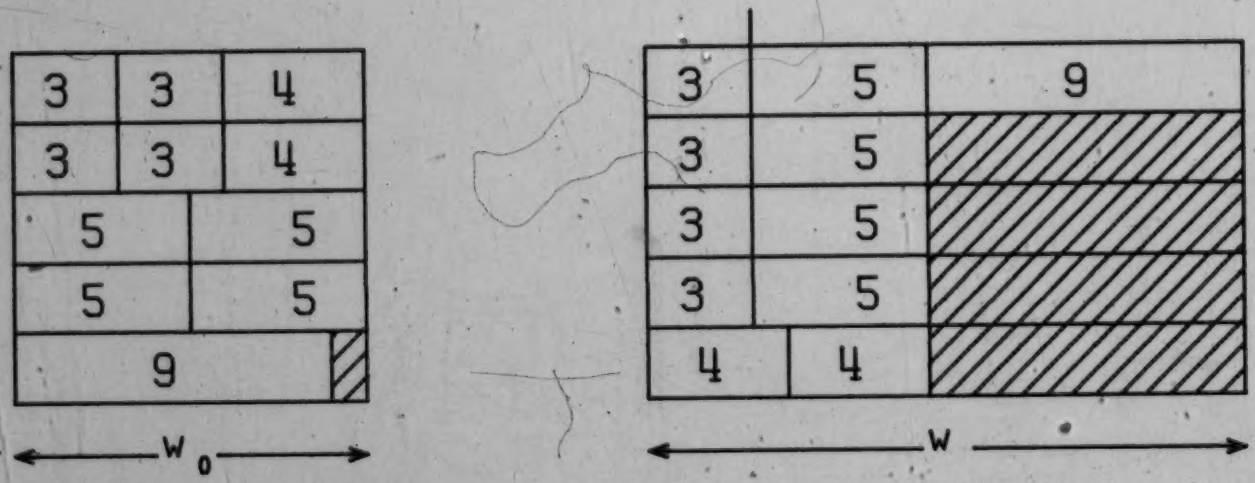
(f) Here, five processors must share twelve tasks. At least two of the processors must have at least three tasks each leaving no more than six tasks for three processors. Again, even if those tasks assigned three per processor take $t_L/3$ each this leaves

$$y \leq (4(t_L + e) - 6t_L/3)/3$$

and $$w/w_o \leq 2 - 1/3.$$

The schedule in Figure 10 shows the bound to be tight.

∎



$$\frac{W}{W_0} = \frac{17}{10}$$

FIGURE 10: Worst case for m=5, r≤3

2.3.2 Tasks With Largest Execution Time Ratio ≤ 2

In this sub-section, the case when $r \leq 2$ is considered. It is first shown by contradiction that $w/w_o \leq 5/3 - 2/(3m)$ and that this bound is best possible for even m. Subsequently, a technique similar to that used in the previous sub-section is used to prove the tight bound $5/3 - 2/(3(m-1))$ for odd m.

Lemma 2.10: If $w/w_o > 5/3 - 2/(3m)$ and $r \leq 2$ then

(i) $t_L > 2w_o/3$.

(ii) no processor in Z_o will execute more than two tasks and the processor with T_L executes no other tasks.

Proof: (i) Rewrite the ratio as $w/w_o > 2 - 1/p$ with $p = 3m/(m+2)$. By Lemma 2.2, $w_o < (p(m-1)t_L)/((p-1)m)$. Substituting p into the inequality gives $t_L > 2w_o/3$.

(ii) If there exists one processor which executes more than two tasks in Z_o or if the processor with T_L executes more than one task, then the length of Z_o is at least $3t_L/2 > w_o$, which is a contradiction. Thus (ii) is obtained for $r \leq 2$. ∎

Theorem 2.3: If $r \leq 2$ and $m \geq 4$, then $w/w_o \leq 5/3 - 2/(3m)$.

Proof: By contradiction. From Lemma 2.1 the tasks on $(m-1)$ processors in Z_o will be executed on m processors in Z. Furthermore, if $w/w_o > 5/3 - 2/(3m)$, it is obvious from

Lemma 2.10 that at least one of the 1-partnered tasks in Z_o will be 0-partnered in Z. As the processing time for the 1-partnered tasks in Z_o is at most $w_o - t_L/2$, it follows that $w \leq w_o - t_L/2 + t_L = w_o + t_L/2$ and $w/w_o \leq 1 + t_L/(2w_o)$, which is less than or equal to $5/3 - 2/(3m)$ for $m \geq 4$. Thus the theorem is proved. ∎

Theorem 2.4: If $r \leq 2$ and $m \leq 4$, then $w/w_o \leq 3/2$.

Proof: By contradiction. Suppose $w/w_o > 3/2$. Then by Lemma 2.2, $w_o < (p(m - 1)t_L)/((p - 1)m)$ and for $p = 2$, $m \leq 4$, this gives $w_o < 3t_L/2$. Hence, no processor executes more than two tasks in Z_o and the processor which executes T_L executes no other tasks. But this implies $w/w_o \leq 3/2$ (similar to the proof in Theorem 2.3) which is a contradiction. ∎

The bounds are best possible by considering the examples given in Figure 11 for $m = 2$, 3 and $2k$, $k \geq 2$.

However, the bound $5/3 - 2/(3m)$ is not tight for odd $m \geq 5$. A tight bound $5/3 - 2/(3(m-1))$ can be proved for odd $m \geq 5$ by arguments similar to those used in Lemmas 2.8 and 2.9 and Theorem 2.1.

Theorem 2.5: If $r \leq 2$ and $m \geq 5$, m odd, then $w/w_o \leq 5/3 - 2/(3(m - 1))$.

Proof: The proof is similar to that of Lemmas 2.8, 2.9 and Theorem 2.1 and is therefore only sketched here.

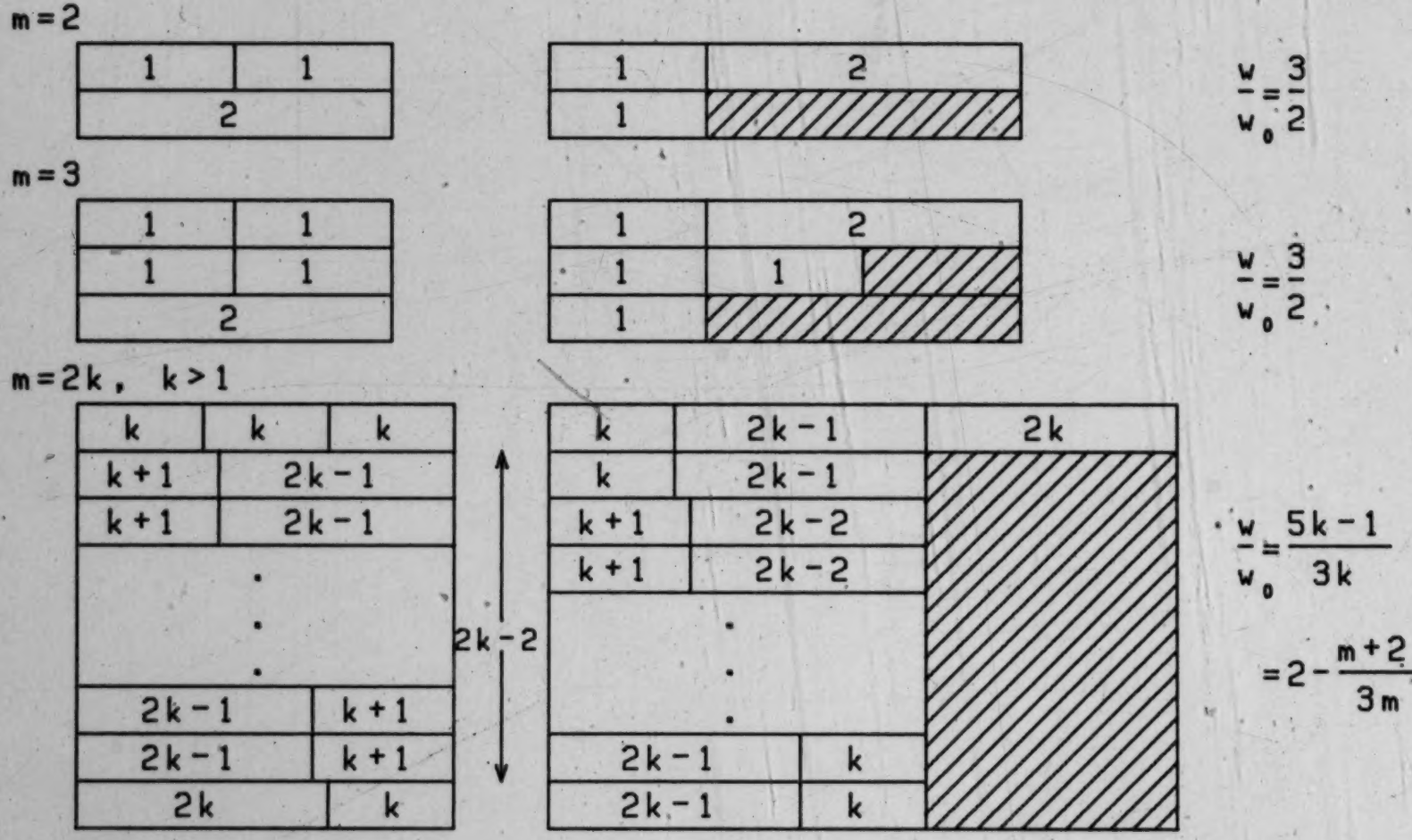


FIGURE 11: Worst cases for m=2,3 and even m>3, r≤2

First note the following points:

(a) Lemma 2.1 applies and hence $w = y + t_L$.

(b) $w_o > 3t_L/2$ for otherwise it can be shown by Lemma 2.10 that $w/w_o \leq 3/2$.

(c) Given counter-example schedules Z and Z_o, $w/w_o > 5/3 - 2/(3(m-1))$ but it has already been proved that $w/w_o \leq 5/3 - 2/(3m)$. Hence, $5/3 - 2/(3(m-1)) < w/w_o \leq 5/3 - 2/(3m)$. Scale the tasks of the counter-example schedules such that $3(m-1)/2 < w_o \leq 3m/2$ and $w/w_o = 5/3 - 1/w_o$. Then $w = 5w_o/3 - 1$, $t_L = 2w_o/3 - g$, for some $g \geq 0$ (see (b) above) and $y = w_o - 1 + g$.

(d) There are no 0-partnered tasks in Z and Z_o, and T_L has a partner in Z_o. First consider schedule Z. If task T_k is 0-partnered then $t_k \geq y$. But $t_k \leq t_L$. Hence, $t_L \geq y$ and $w = y + t_L \leq 2t_L$. By (b) above, $w/w_o \leq 2(2w_o/3)/w_o = 4/3$ which is less than or equal to $5/3 - 2/(3(m-1))$ for $m > 3$, giving a contradiction.

For schedule Z_o, if T_L has no partner then total execution time of the remaining $m - 1$ processors in Z_o is $(m - 1)w_o$. Hence in scheduling the tasks in Z, y becomes at most $(m - 1)w_o/m$. Therefore $w/w_o = (y + t_L)/w_o$ is at most $((m - 1)w_o/m + t_L)/w_o \leq (m-1)/m + 2/3$, (by (b)). This is a contradiction since $(m - 1)/m + 2/3$ is less than $5/3 - 2/(3(m - 1))$ for $m > 3$. The case for any other

task not having a partner in Z_o can be obtained by a similar consideration of total busy periods of all processors.

(e) There is at least one processor in Z_o which performs three tasks. Otherwise, as in the proof of Theorem 2.4, $w/w_o \leq 3/2$.

(f) Given counter-example schedules Z, Z_o, either the following hold or another pair of schedules Z', Z'_o can be constructed for which they hold.

(i) Any 2-partnered task (or T_L's partner) is smaller than any 1-partnered task.

(ii) If a 1-partnered task T_k is larger than or equal to its partner then $t_k \geq w_o/2$. The proof only requires arguments similar to those of Lemma 2.6.

(g) The 1-partnered tasks can be ordered in both Z and Z_o by Lemma 2.7.

The above facts lead to the configuraton in Figure 12 for a counter-example. A processor reduction by two is done by deleting tasks $\{T_{i+u+1},\ldots T_{i+u+4}\}$. This is followed by reducing task times thus:

$$t'_j = \begin{cases} t_j - 1, & j \leq i+u, \\ t_j - 2, & j \geq i+u+5, \end{cases}$$

which leads to a counter-example for $m - 2$ processors.

For the initial case, $m = 5$, the proposed bound $5/3 - 2/(3(m-1))$ is equal to $3/2$. All preprocessing of counter-examples before the processor reduction stage are

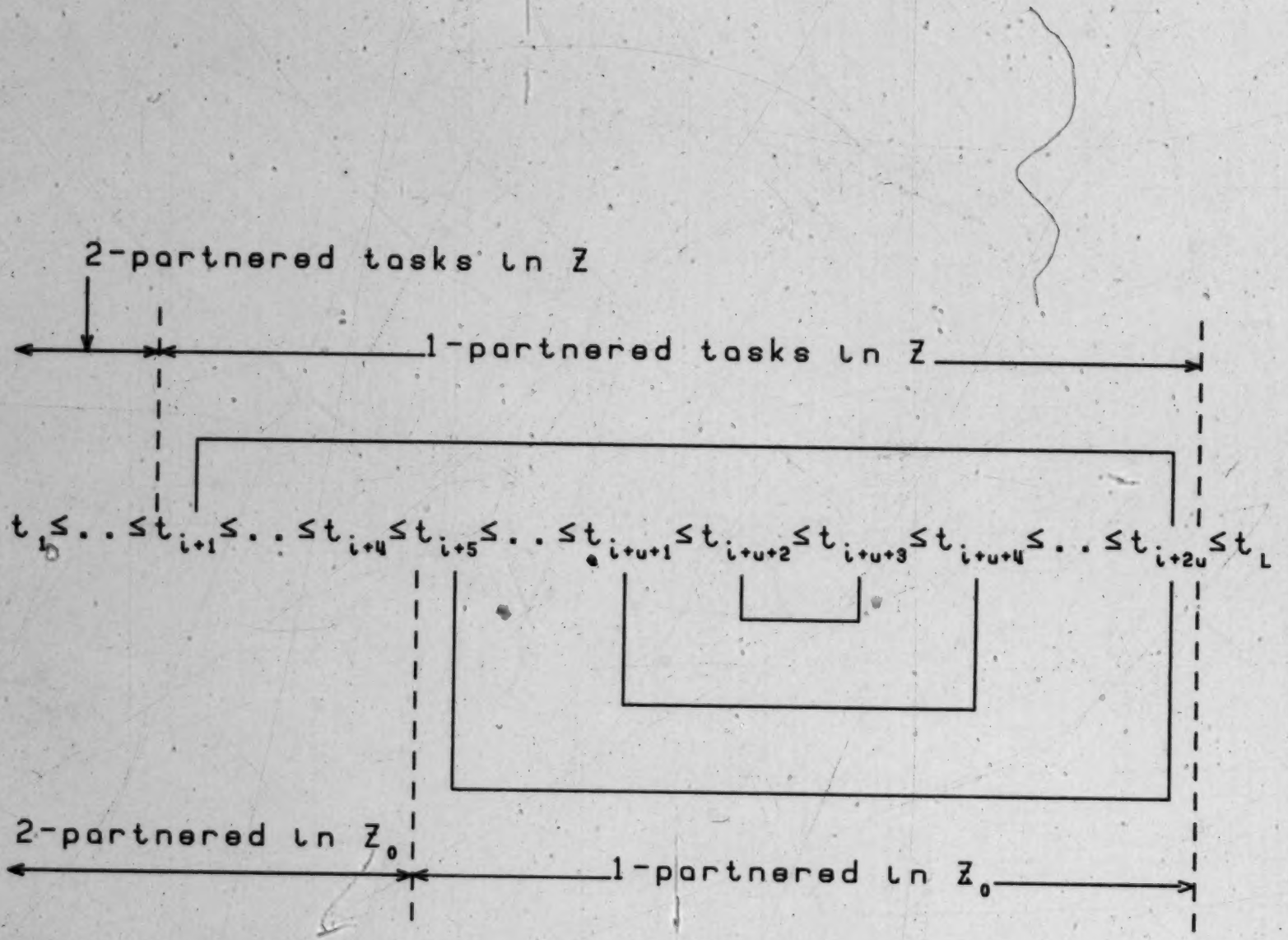

2-partnered tasks in Z
1-partnered tasks in Z
$t_1 \le .. \le t_{i+1} \le .. \le t_{i+4} \le t_{i+5} \le .. \le t_{i+u+1} \le t_{i+u+2} \le t_{i+u+3} \le t_{i+u+4} \le .. \le t_{i+2u} \le t_L$
2-partnered in Z_0
1-partnered in Z_0

FIGURE 12: Order of tasks in normalized counter-example, $r \le 2$

found to be valid for m = 5. Thus (see Figure 12) one obtains a pair of tasks T_i, T_j (1-partners in Z), for which T_i is 2-partnered in Z_o, T_j is 1-partnered in Z_o and is smaller than or equal to its partner in Z_o. Now $t_i \leq w_o - t_L$ since its two partners must take at least $t_L/2$ each, and $t_j \leq w_o/2$.

Hence, $$y \leq t_i + t_j \leq w_o - t_L + w_o/2$$

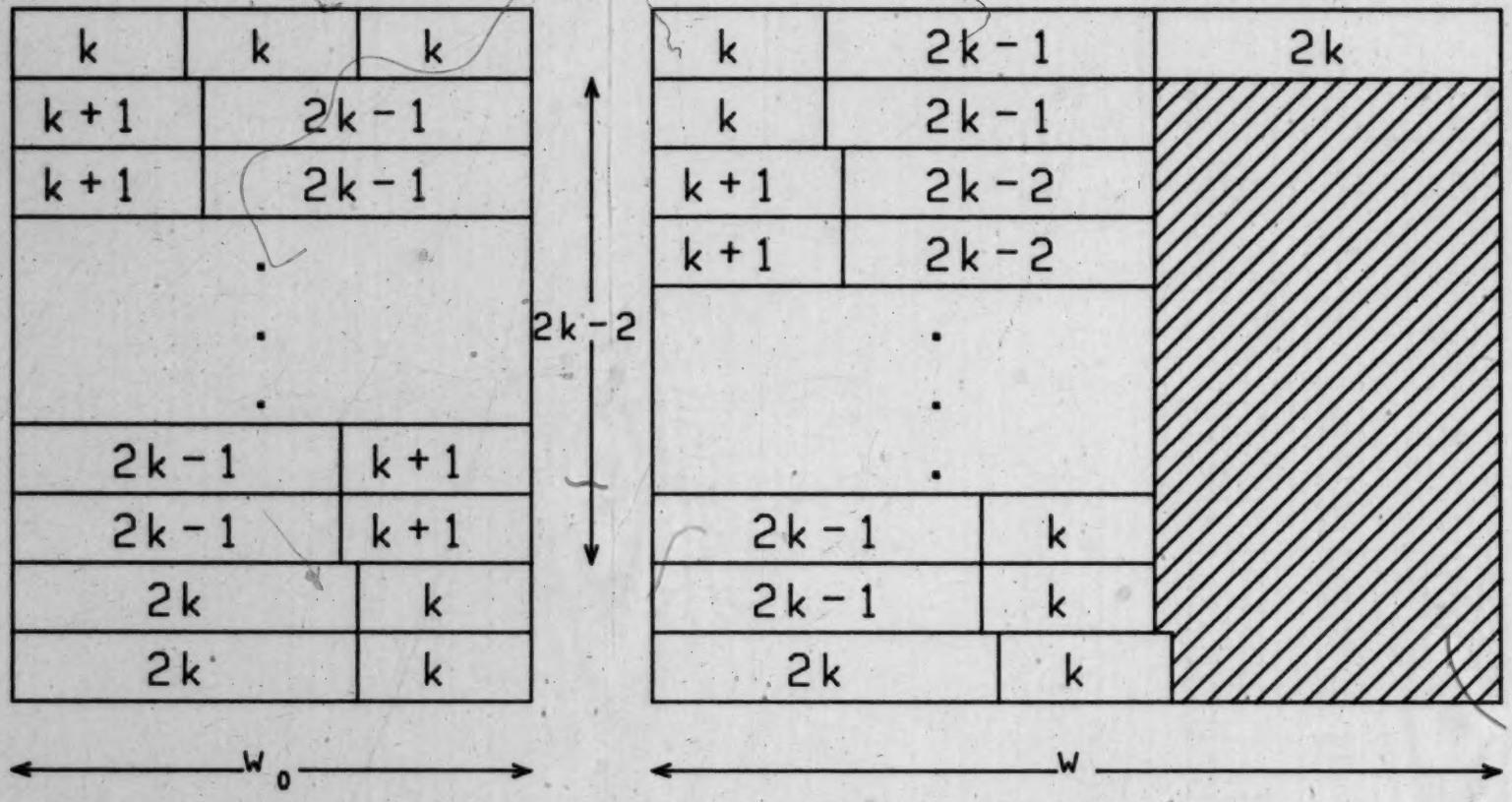
and $$w/w_o \leq 3/2.$$

The bound is tight by Figure 13. ∎

2.4 Discussion

This chapter has investigated the behaviour of tight bounds on list schedules of independent tasks on m identical processors as the degree of similarity between the tasks' execution times is varied. Out of this investigation the relationships in Figure 14 emerge. As might be intuitively expected, the worst case heuristic schedule length approaches the optimal as the ratio r is reduced. Note however that even for a ratio as low as 2, the heuristic can still take up to 3/2 times as long as the optimal.

Another interesting point to note is that the examples which illustrate the tightness of the bounds all attain the maximum ratio allowed, an indication that the bounds might be tightened further for values of r in between the integer values considered.



$$m = 2k+1, \quad k > 1; \quad \frac{w}{w_0} = \frac{5k-1}{3k} = 2 - \frac{m+1}{3m-3}$$

FIGURE 13: Worst cases for odd $m > 4$, $r \leq 2$

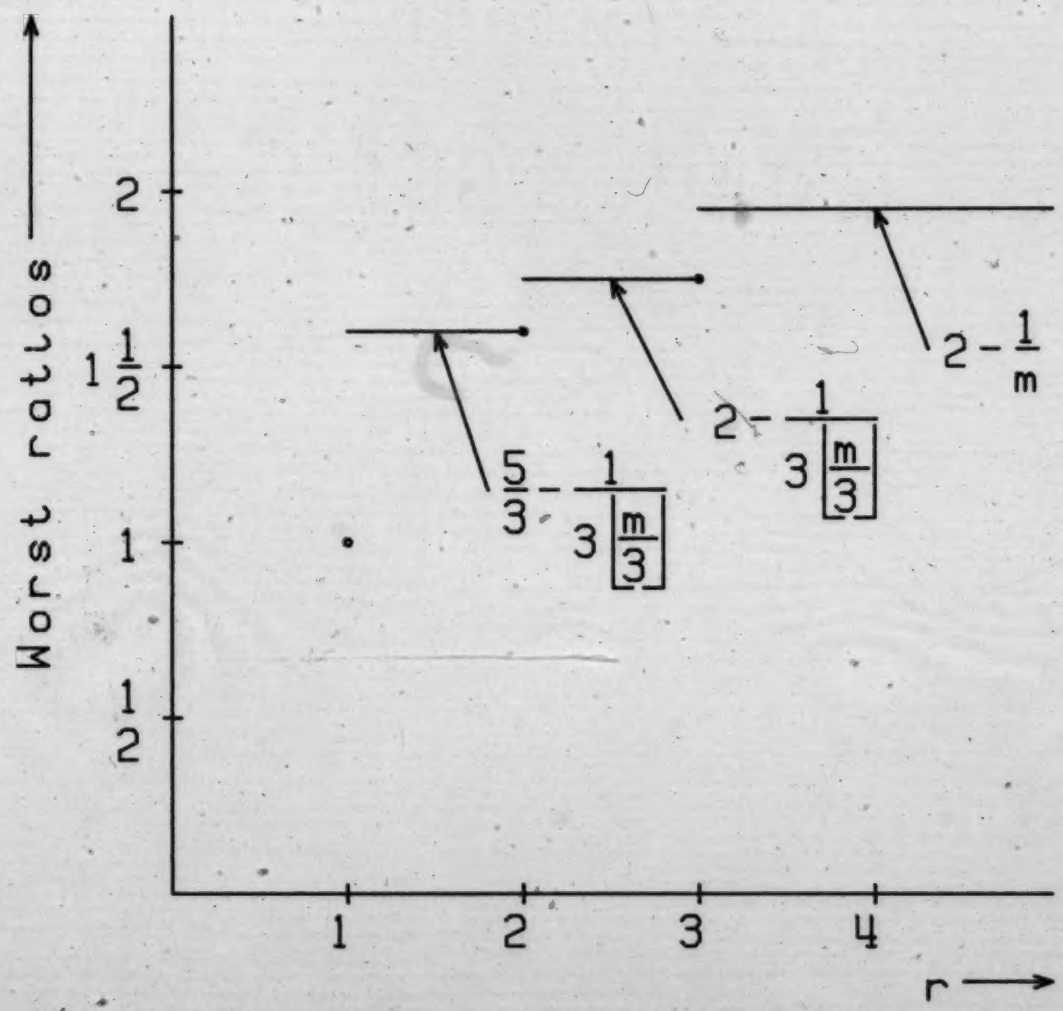


FIGURE 14: Worst ratios vs r

Chapter Three

PROCESSOR BOUND SYSTEMS

Consider the problem of scheduling a set of n tasks, $\{T_1, T_2, \ldots, T_n\}$ on a multiprocessor computer system that has $m \geq 2$ processors $\{P_1, P_2, \ldots, P_m\}$ capable of independent operation on independent tasks. As usual, it is assumed that there is a partial ordering, $<$, specified on the set of tasks in the form of a directed acyclic graph. In addition, it is also assumed that no two tasks are identical. Thus, for each task T_i a processor index, $R(T_i)$, is specified so that task T_i must be executed on the $R(T_i)$-th processor. Such a system is called an m-processor bound system. This model covers such well known instances of processor-bound systems such as the flow and job shops (Johnson, 1954; Garey, Johnson & Sethi, 1976; Gonzalez & Sahni, 1978; Chin & Tsai, 1978) and the open shop systems (Gonzalez & Sahni, 1976; Gonzalez, 1976).

In this chapter, non-preemptive schedules for the case in which all tasks have the same execution time (unit execution time or UET) is considered. It is shown that the problem of scheduling such systems to minimize schedule length is NP-complete even when the tasks have a very simple precedence structure consisting of chains. Subsequently, a dynamic programming solution is presented.

3.1 Survey

Goyal (1977) shows that the problem of scheduling m-processor bound UET systems to minimize schedule length is NP-complete in two restricted cases:

(1) m-processor bound UET systems, arbitrary m, with the precedence constraints restricted to being a forest,

(2) 2-processor bound systems with arbitrary precedence constraints.

The first is shown by a reduction from the NP-complete problem of Node Covering (Garey & Johnson, 1975; Karp, 1972) and the second by a reduction from the UET scheduling problem (Ullman, 1974).

Furthermore, Goyal presents a simple level algorithm, similar to that of Hu (1961), which produces optimal schedules if the precedence graph is in the form of a cyclic forest, that is, one in which all tasks in the same level of the forest require the same processor, but for any two tasks T_i and T_j in two adjacent levels h and h-1 respectively, their processor requirements satisfy the relation

$$R(T_i) = \begin{cases} R(T_j)+1, & 1 \leq R(T_j) < m, \\ 1, & R(T_j) = m. \end{cases}$$

Thus, the problem of scheduling m-processor bound UET systems with a specific value of m and with the precedence relations restricted to being a forest is left open.

Liu and Liu (1977) and Jaffe (1978) consider a slightly

more general model in which there are q types of processors, with m_i identical processors of type i, $1 \leq i \leq q$, and derive bounds on the lengths of list schedules in terms of the optimal schedule length.

3.2 Definitions

If the precedence constraints include the relation $a < b$ for tasks a and b, then a is an immediate predecessor of b and b is an immediate successor of a. The predecessors of b are all those tasks which must be done before b can be executed (and thus includes a and its predecessors). Similarly, the successors of task a are all tasks which can be done only after task a has been executed (and thus includes b and its successors). The precedence graph consists of chains if each node has at most one immediate predecessor and at most one immediate successor. It is a terminally rooted tree if each node has at most one immediate successor and there is exactly one node, the root, which has no successor. Similarly, it is an initially rooted tree if each node has at most one immediate predecessor and there is exactly one node, the root, which has no predecessor. Nodes which have no predecessors (successors) in a terminally (initially) rooted tree are called leaf nodes. A terminally rooted forest consists of a set of terminally rooted trees. Similarly, an initially rooted forest consists of a set of initially rooted trees.

An m-processor bound UET scheduling problem with precedence constraints restricted to k chains will be referred to as a k-chain problem. Similarly, an m-processor bound UET scheduling problem with a precedence constraint which is a terminally rooted tree will be referred to as a tree problem.

3.3 Complexity of the k-Chain Problem

In this section, it is shown that the k-chain problem for arbitrary k is NP-complete. The proof is by reduction from 3-PARTITION (see Section 1.2.2).

Theorem 3.1: The k-chain problem for arbitrary k is NP-complete.

Proof: Given an instance of 3-PARTITION consider the following (3n+1)-chain problem. The chains are Q_i, $0 \le i \le 3n$. The j-th task of the i-th chain is $Q_i[j]$. Chain Q_0 has 2nK tasks while chain Q_i, $1 \le i \le 3n$, has $2a_i$ tasks. The processor requirements are

$$R(Q_0[j]) = \begin{cases} 2 & \text{for } (2x-2)K+1 \le j \le (2x-1)K, \\ 1 & \text{for } (2x-1)K+1 \le j \le 2xK, \quad 1 \le x \le n, \end{cases}$$

$$R(Q_i[j]) = \begin{cases} 1 & \text{for } 1 \le j \le a_i \\ 2 & \text{for } a_i < j \le 2a_i. \end{cases}$$

The deadline is D = 2nK, the length of chain Q_0. Consequently, any schedule that meets the deadline must execute a task of chain Q_0 in each time interval. This gives

the template of Figure 15.

Suppose the 3-PARTITION problem has a solution. Schedule the processor-1 tasks of those chains corresponding to the three elements of the i-th partition in the i-th idle slot (left by chain Q_0) on the first processor and the remaining processor-2 tasks of the same three chains on the i-th slot on the second processor.

Conversely, suppose the chain problem has a schedule which finishes by time D. Then, the schedule must have no idle periods. Chain Q_0 must be executed as shown in Figure 15, leaving 2n idle periods, each of length K, alternating between the processors.

It is easily shown by induction on the number, n, of idle periods on either processor that for chains Q_i, $i > 0$, if the first task of chain Q_i is done in time slot $[(2x-2)K+1, (2x-1)K]$, then the last task of the same chain must be executed in time slot $[(2x-1)K+1, 2xK]$, $1 \leq x \leq n$. Hence, all the processor-1 tasks of Q_i are done in one of the idle periods of Figure 15 and all the processor-2 tasks in the next idle period.

In general, let y_x be the number of chains Q_i, $i > 0$, whose processor-1 tasks are done in time slot $[(2x-2)K+1,(2x-1)K]$, $1 \leq x \leq n$. Then, by the above result, the processor-2 tasks of the same chains are done in time slot $[(2x-1)K+1,2xK]$. It follows that the total number of

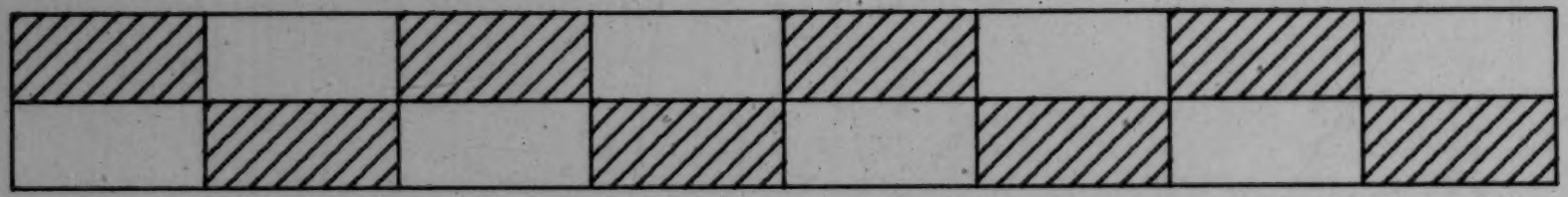


FIGURE 15: Template of Theorem 3.1, n=4.
Idle periods are shaded.

processor-1 (or 2) tasks from the y_x chains is K. Since $K/4 < a_i < K/2$, y_x must be precisely three for any x, $1 \leq x \leq n$. Hence, the 3-PARTITION problem has a solution. ∎

3.4 Solution of the 2-Chain Problem

Let the tasks in one of the two chains be a_i, $1 \leq i \leq s$, and the tasks in the other chain, b_j, $1 \leq j \leq t$, with precedence relations $a_1 < a_2 < \ldots < a_s$ and $b_1 < b_2 < \ldots < b_t$, where $n = s+t$ is the total number of tasks. There are $m \geq 2$ processors.

The solution is by dynamic programming. The principle of optimality is illustrated in the following argument. In order to obtain an optimal schedule one can proceed as follows. If $R(a_1) \neq R(b_1)$ then a_1 and b_1 can be executed in the first time unit without loss of optimality. This is then followed by an optimal schedule of the sub-chains $a_2 < \ldots < a_s$ and $b_2 < \ldots < b_t$. If, on the other hand, $R(a_1) = R(b_1)$ then only one of $\{a_1, b_1\}$ can be executed in the first time interval. In one case a_1 is followed by an optimal schedule of the sub-chains $a_2 < \ldots < a_s$ and $b_1 < \ldots < b_t$, while in the other case, b_1 is followed by an optimal schedule of the sub-chains $a_1 < \ldots < a_s$ and $b_2 < \ldots < b_t$. For the optimal solution, try both ways and pick the better one.

Let $F(i,j)$, $1 \leq i \leq s$, $1 \leq j \leq t$, be the length of an optimal schedule for the sub-chains $a_i < \ldots < a_s$ and $b_j < \ldots < b_t$. Then the above discussion implies the relation

$$F(1,1) = \begin{cases} 1 + F(2,2), & \text{for } R(a_1) \neq R(b_1) \\ 1 + \text{MIN}\{F(2,1), F(1,2)\}, & \text{for } R(a_1) = R(b_1). \end{cases}$$

In general, assume that the first i-1 tasks of the a-chain and the first j-1 tasks of the b-chain have been scheduled without loss of optimality. (A number of tasks is said to have been scheduled without loss of optimality in the first r time units, r > 0, if there exists an optimal schedule with identical execution for the r units.) Then, the above argument can be repeated for tasks a_i and b_j and the following recursion obtained.

$$F(i,j) = \begin{cases} 1 + F(i+1,j+1), & \text{for } R(a_i) \neq R(b_j) \\ 1 + \text{MIN}\{F(i+1,j), F(i,j+1)\}, & \text{for } R(a_i) = R(b_j). \end{cases}$$

Since chain $b_j < \ldots < b_t$ is not defined for $j > t$, let $F(i,t+1)$ denote the length of an optimal schedule (equal to length of sub-chain) of the sub-chain $a_i < \ldots < a_s$, for $i \leq s$. Similarly, let $F(s+1,j)$ be the length of the sub-chain $b_j < \ldots < b_t$, for $j \leq t$. Thus, $F(i,t+1) = s+1-i$, $i \leq s$, and $F(s+1,j) = t+1-j$, $j \leq t$. Taking $F(s+1,t+1)$ to be zero, every element of the array $F(i,j)$, $1 \leq i \leq s$, $1 \leq j \leq t$, can easily be computed. To perform the computation efficiently, it is essential to compute $F(i+1,j)$, $F(i,j+1)$ and $F(i+1,j+1)$ before ever attempting to compute or use the value of $F(i,j)$. One way to achieve this is to compute the elements of $F(i,j)$ in reverse row order i.e. last row first and for each row, last column first. This computation is presented in Procedure 1.

Theorem 3.2: An optimal schedule for the 2-chain

```
   procedure CHAIN-ALG;
1. begin comment chains are a_1<...<a_s and b_1<...<b_t.
        F(i,j) is length of optimal schedule for
        sub-chains a_i<...<a_s and b_j<...<b_t, where a
        sub-chain is empty if i>s or j>t respectively.
        This procedure computes the array F(i,j) in
        reverse row order;
2. for i := 1 until s+1 do F(i,t+1) := s+1-i;
3. for j := 1 until t do F(s+1,j) := t+1-j;
4. for i := s step -1 until 1 do
5.     for j := t step -1 until 1 do
6.         begin if R(a_i) ≠ R(b_j) then
7.             F(i,j) := 1 + F(i+1,j+1) else
8.             F(i,j) := 1 + MIN(F(i+1,j),F(i,j+1));
9.         end;
10. end;
```

PROCEDURE 1

problem can be constructed in O(st) time and space using the procedure CHAIN-ALG.

Proof: Suppose the array F(i,j) has been computed with the given procedure. Construct a schedule having minimal length F(1,1) by tracing the computation of F(1,1).

Suppose tracing is currently at F(i,j). If $R(a_i) \neq R(b_j)$ schedule a_i and b_j in the next time unit and trace F(i+1,j+1). If $R(a_i) = R(b_j)$ then schedule a_i and trace F(i+1,j) if F(i,j) = 1 + F(i+1,j); otherwise schedule b_j and trace F(i,j+1). It is clear from the foregoing discussion that this will result in a schedule with length F(1,1). This process takes O(F(1,1)) = O(s+t) time.

Now, the array F(i,j), $1 \leq i \leq s+1$, $1 \leq j \leq t+1$, has O(st)

elements and the computation of each element (see Procedure 1) is done within a constant number of steps. Hence, at most O(st) time is used to find an optimal schedule. The O(st) space requirement is obvious. ∎

Example 3.1: Let m = 3, s = 6, t = 7,

$a_i = i$, $1 \le i \le 6$, $b_j = j$, $1 \le j \le 7$,

$(R(a_i), 1 \le i \le 6) = (1, 1, 3, 3, 2, 2)$,

$(R(b_j), 1 \le j \le 7) = (1, 3, 3, 1, 2, 2, 1)$.

Then the elements of array F(i,j) are as follows:

index i

↓	1	2	3	4	5	6	7	8 ← index j
1	8	7	7	7	6	6	6	6
2	8	7	6	6	5	5	5	5
3	8	7	6	5	4	4	4	4
4	7	7	6	5	4	3	3	3
5	7	6	5	4	4	3	2	2
6	7	6	5	4	3	2	1	1
7	7	6	5	4	3	2	1	0

The minimal length is F(1,1) = 8 units. One of the possible tracings is -->F(1,1) -->F(1,2) -->F(2,3) -->F(3,4) -->F(4,5) -->F(5,6) -->F(6,6) -->F(6,7). The schedule which is generated in conjunction with the above tracing is given in Figure 16. ∎

An alternative way to recover an optimal schedule from the F(i,j) array is to keep another array H(i,j) of pointers indicating which of F(i+1,j), F(i,j+1), and F(i+1,j+1) was used to obtain F(i,j). This simplifies the subsequent task of constructing an optimal schedule after computation of

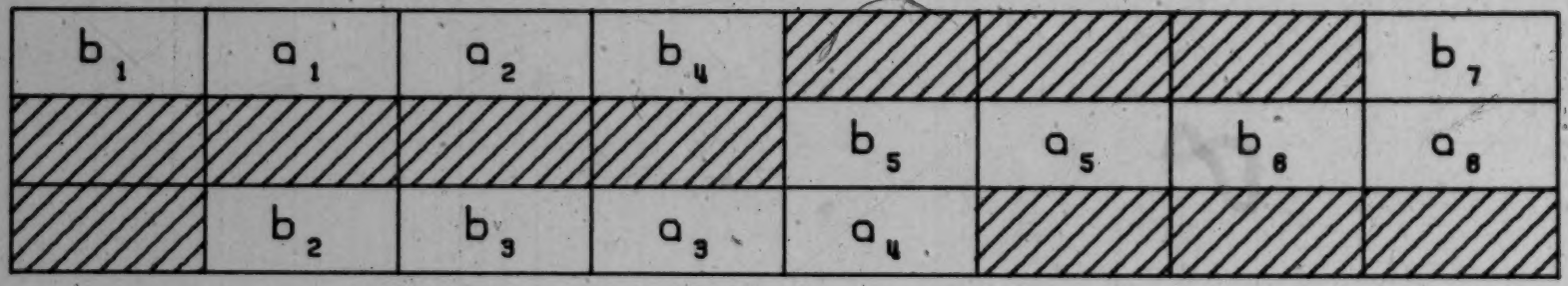


FIGURE 16: Optimal schedule of Example 3.1

F(1,1).

The above algorithm is strikingly similar to the dynamic programming solution of the longest common subsequence (LCS) problem (Hirschberg, 1975; Brown, 1978). In fact, for a 2-chain problem on two processors, if the processor requirements of one of the chains are reversed (that is, processor-1 tasks become processor-2 tasks and vice versa) a solution of the LCS problem for the resulting chains corresponds to a solution of the original scheduling problem. Thus, the 2-chain problem on two processors is equivalent to an LCS problem with a 2-symbol alphabet. Considerable work has been done on the LCS problem (see Brown for further references) which may be applicable to the 2-chain problem. Unfortunately, the analogy breaks down for the k-chain problem on m processors when $k > 2$ or $m > 2$.

3.5 Extension to More Complex Precedence Graphs

In this section, the extension of the dynamic programming solution to k-chains, trees, and arbitrary precedence digraphs is considered. The extension to k chains, $k > 2$, is straightforward, the two-dimensional array, F, of the previous section being replaced by a k-dimensional array. This leads to an $O(n^k)$ time and space algorithm, assuming each chain to be of length n. Now consider the tree problem for $m = 2$. (The extension of the solution to the case of more than two processors will be

straightforward).

For an arbitrary precedence graph (a directed acyclic graph) a subgraph, G, will be called a terminal subgraph if it satisfies the condition that if a node, u, is in G then every successor of u is also in G. A terminal subtree is a terminal subgraph which is also a tree. The set of nodes in a terminal subgraph forms a terminal subset of the set of nodes in the original graph.

Now, consider the tree problem. Given a schedule for the tree problem, for any integer, t, (less than or equal to the length of the schedule) the tasks in the final t time units of the schedule form a terminal subset of the set of all tasks in the system. The principle of optimality applies in the same manner as for the 2-chain problem. Suppose that at the end of the (i-1)-th time unit, a number of tasks have been scheduled without loss of optimality, leaving a terminal subtree, G. The only nodes that can be scheduled for execution in the i-th time unit are leaf nodes of G. Consider all possible pairings of a processor-1 leaf, g_1, of G with a processor-2 leaf, g_2, of G for execution in time unit i. From each such pairing followed by an optimal schedule of the corresponding reduced terminal subtree, pick the one which gives the shortest length schedule. This leads to the recursion, $f(G) = MIN\{1 + f(G - g_1 - g_2)\}$, where f(G) is the length of an optimal schedule for subtree G and the minimum is taken over all leaves g_1 of G which require the

first processor and all leaves g_2 of G which require the second processor. Of course, G may have no processor-1 leaf or processor-2 leaf in which case $(G - g_1 - g_2)$ in the expression above is replaced by $(G - g_2)$ or $(G - g_1)$. In order to compute f(G) efficiently for all terminal subtrees, G, the following is required:

(a) A method of indexing or assigning addresses to subtrees G so that the locations of $(G - g_1 - g_2)$, $(G - g_1)$ and $(G - g_2)$ can be referenced quickly given G, g_1 and g_2,

(b) A simple method of enumerating the terminal subtrees G (or equivalently, enumerating the terminal subsets) such that $(G - g_1 - g_2)$, $(G - g_1)$ and $(G - g_2)$ are enumerated before G.

The first problem can be solved by assigning labels to the tasks so that the index of a tree is the sum of the labels of the tasks in that tree.

As for the second problem, although several algorithms are available (Nijenhuis & Wilf, 1978) for enumerating subsets of a set under varying conditions, none of them can be used to enumerate terminal subsets exclusively. In the following, an enumeration algorithm is presented which enumerates only the terminal subsets of a given tree. This reduces storage space for indexing and eliminates the need to check that a subset is terminal or alternatively the need to derive f(G) for a subtree G that would never subsequently be referenced.

The terminal subset enumeration scheme consists of two parts, a labelling procedure, LABELTREE, which is presented in Procedure 2 and a decoding procedure, DECODE, given in Procedure 3. LABELTREE assigns labels to the nodes of a tree so that each terminal subset may be indexed by the sum of the labels of the nodes in the subset. It will be shown that the terminal subsets have indices from 1 to I, where I is the sum of all the labels. DECODE(j) constructs the terminal subset whose index is j. Thus in order to enumerate the terminal subsets it is sufficient to DECODE(j) for $1 \leq j \leq I$.

The labelling procedure groups the tasks into N chains and assigns the same label to all tasks on the same chain. Assume that a dummy final node (successor to the root) with label 0 is temporarily added to the tree and that the predecessors of each node have been arbitrarily ordered. Then, the iterative step proceeds as follows. Suppose the first (i-1) chains have been defined and labelled. Then, define the i-th chain in the following manner:

(1) Find the most recently labelled node with an unlabelled predecessor. If none exists, stop (all tasks have been labelled). Add the next unlabelled predecessor of the node to the new chain. (It is the last task on the chain).

(2) Let v be the most recent node added to the chain. If v has any predecessors (none of them has a label) add the first predecessor of v to the chain and go to 2;

```
    procedure LABELTREE;
    begin comment N is the number of chains.
              I is the index of the complete tree;
1.  i := a1 := 1; x := t := n1 := s1 := 0;

2.  TRAVERSE(root of tree, x);
3.  N := i; I := t;
    end.

    Procedure TRAVERSE(node, x);
    begin  local integer k;
1.  if null node return;
    comment visit node;
2.  label node with ai; add node to chain;

    x := x + ai;    t := t + ai;    ni := ni + 1;

    comment traverse predecessors in pre-order;
3.  TRAVERSE(first predecessor of node, x);
    k := (number of predecessors of node) - 1;
4.  while k > 0 do
        begin comment start new chain;
        i := i + 1; ni := 0; si := x;

        ai := t + 1 - x;

        TRAVERSE(next predecessor of node, x);
        k := k - 1;
        end;
    end;
```

PROCEDURE 2

otherwise stop (chain i is complete).

This chain becomes the i-th chain. The label, a_i, assigned to every task on this chain is one larger than the sum of labels of all those labelled tasks that are not successors of the last task on the chain. The sum, s_i, of labels of all labelled tasks that are successors of the last task on the chain is also saved for later use in the decoding process. With this labelling, the index of a set will be the sum of the labels of tasks in that set thus satisfying condition

(a) above for efficient computation of f(G).

The procedure, as given, implements the above ideas using a preorder traversal (Aho, Hopcroft & Ullman, 1974) of the tree. The procedure keeps track of the sum, x, of the labels of all labelled tasks that are successors of the task to be visited and the total, t, of labels of all labelled nodes. These are used in computing a_i and s_i when a new chain is defined.

Given an index, j, the decoding procedure determines the nodes of a unique subtree G of the original set of nodes. It uses the following data set up by the labelling procedure:

(1) the number of chains, N,

(2) the number of tasks, n_i, on chain i,

(3) the actual tasks on each chain in order,

(4) the label, a_i, of tasks on chain i,

(5) the sum of labels, s_i, defined above.

The procedure considers the chains in decreasing order of their index i.e. in the reverse order from that in which they were defined. Suppose chains N,N-1,...,i+1 have been considered i.e. the tasks from these chains have been determined and their labels have been subtracted from j giving current subset index c. Then one of the following cases applies.

(1) $c < s_i + a_i$. Then $k = \emptyset$ tasks of chain i belong to the subset.

```
    procedure DECODE(j);
    begin  comment N is the number of chains.
         The arrays a_i, s_i, and n_i, 1≤i≤N, are as defined
         in Procedure 2;
1.  c := j; SUBSET := null;
2.  for i := N step -1 until 1 do
3.       begin  comment find elements of chain i belonging
                to the subset;
         if c < s_i + a_i then k := 0 else
              if c ≥ s_i + n_i a_i then k := n_i else find k
                   such that s_i + ka_i ≤ c < s_i + (k+1)a_i;
         add last k tasks of chain i to SUBSET;
         c := c - ka_i;
         end;
4.  return SUBSET;
    end.
```

PROCEDURE 3

(2) $c \geq s_i + n_i a_i$. Then $k = n_i$ tasks of chain i belong to the subset.

(3) $s_i + ka_i \leq c < s_i + (k+1)a_i$. (i.e. $k = \lfloor (c - s_i)/a_i \rfloor$.) Then last k tasks of chain i belong to the subset.

In any case, c is reduced by ka_i and the procedure considers chain i-1.

The labelling and subsequent decoding of a particular index j is illustrated by the following example.

Example 3.2:

LABELTREE: See Figure 17 for tree structure. The ordering of predecessors to a node is indicated in the figure by integers near the edges leading into each node.

i	chain i	n_i	a_i	s_i
1	r,d,a	3	1	0
2	z	1	2	2
3	e,b	2	4	2
4	h,f	2	13	1
5	g	1	26	14

DECODE (17):

c = 17; SUBSET is empty

i=5: $c < s_5 + a_5$. (no task from chain 5)

i=4: $s_4 + a_4 \leq c < s_4 + 2a_4$.

(take last task of chain 4)

SUBSET = {h}. $c = c - a_4 = 4$.

i=3: $c < s_3 + a_3$. (no task from chain 3)

i=2: $c \geq s_2 + n_2a_2$. (take all chain 2 tasks)

SUBSET = {h,z}; $c = c - n_2a_2 = 2$.

i=1: $s_1 + 2a_1 \leq c < s_1 + 3a_1$.

(take last two tasks of chain 1)

SUBSET = {h,z,d,r}; $c = c - 2a_1 = 0$. ∎

The following properties of LABELTREE and DECODE are required for proving that the terminal subsets are indeed given by DECODE(j) for $1 \leq j \leq I$.

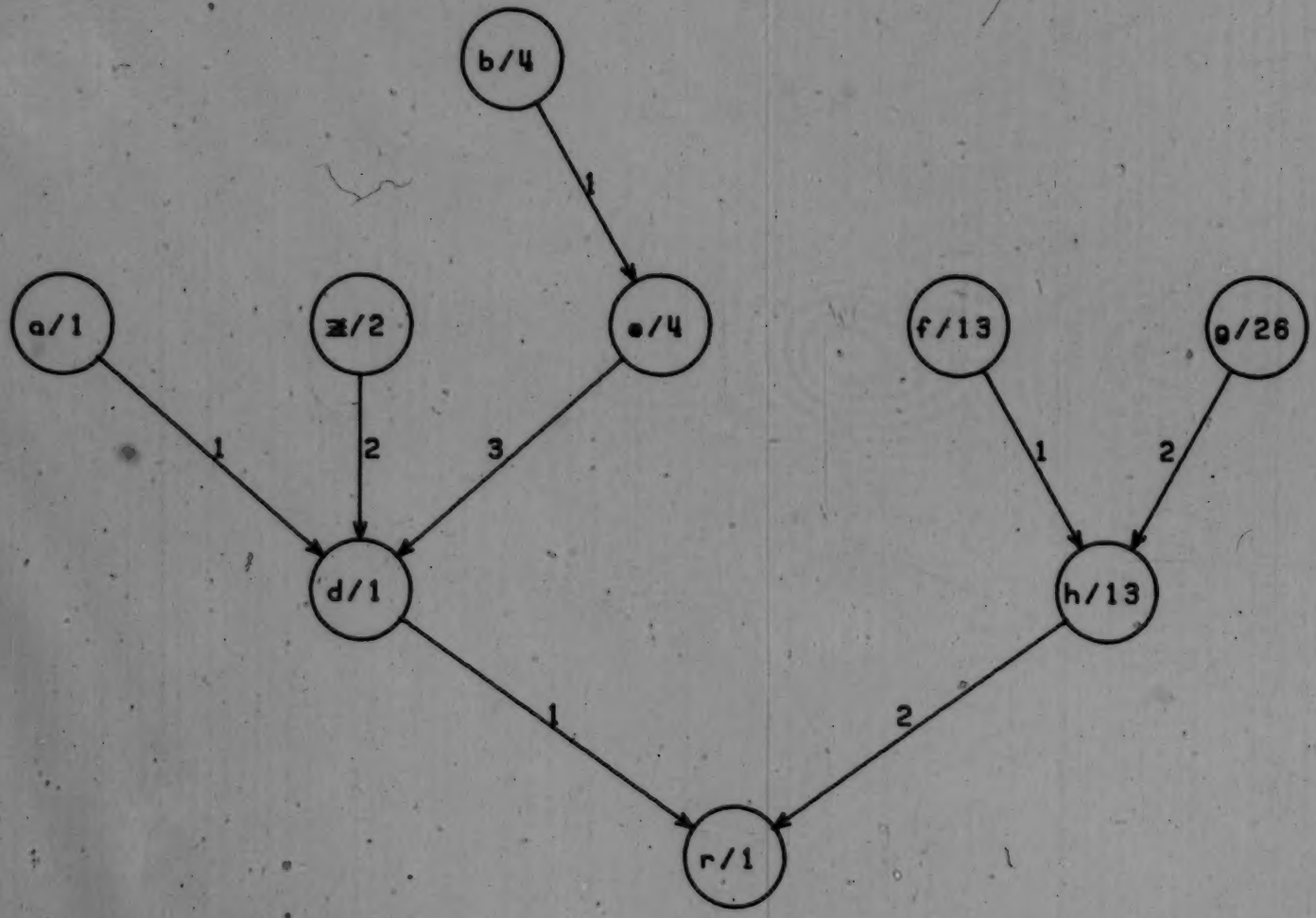


FIGURE 17: Tree of Example 3.2

Lemma 3.1: With the labelling of procedure LABELTREE every terminal subset has a unique index j, $1 \leq j \leq I$.

Proof: The proof is by contradiction. Note that the index of a set is the sum of the labels of its elements. Since I is by definition the sum of all the labels, any set must have index in the given range. Suppose two terminal subsets U and V have the same index j. Let node u, with label a_i, be the highest labelled node which is in one subset but not in the other. Assume that node u is in U but not in V. Then all labels in V which are less than a_i were assigned by LABELTREE before a_i. By definition of a_i in the procedure, a_i is larger than the sum of all smaller labels in the tree which are not labels of successors of node u. Thus, the following situation results:

(1) Every node in V with label larger than a_i is in U.

(2) Node u, with label a_i, is in U but not in V.

(3) Every successor of u in V is also in U since U is a terminal subset and must have all successors of node u.

(4) The sum of labels of nodes in V whose labels are less than a_i and which are not successors of u is strictly less than a_i.

Hence, set U has a higher index than set V, contradiction. ∎

Lemma 3.2: The set J = DECODE(j) identified by any index j is unique for $1 \leq j \leq I$.

Proof: Consider the way in which j is decoded. In the general step the tasks from chains N, N-1, ... i+1 have been determined and their labels subtracted from j giving an index value c for the remaining elements. Now, suppose that no task of chain i is in the set J. Then, J can contain (among the remaining chains) at most all tasks with label less than a_i. The labelling procedure guarantees that the sum of labels of these tasks is at most $s_i + a_i - 1$. Hence, $c < s_i + a_i$. Similarly, suppose that k tasks of chain i belong to set J. Then J has sum of labels at least $s_i + ka_i$ for the k tasks and the successors of the last task of chain i which must all be in the set. In addition, J may contain all tasks with label less than a_i which are not successors of the last task of chain i. Since this latter set of tasks has total labels at most a_i-1, $s_i+ka_i \leq c < s_i+(k+1)a_i$. In the case that all n_i tasks of chain i are in the set, the inequality $s_i + n_ia_i \leq c$ is similarly obtained.

Since only one of the three cases

(1) $c > s_i + a_i$,

(2) $s_i + ka_i \leq c < s_i + (k+1)a_i$

(3) $c \geq s_i + n_ia_i$

can occur for a given c, the value of c uniquely identifies the number of tasks from chain i. Thus, DECODE(j) produces a unique subset for index j. ∎

It is also necessary to show that I is actually the number of non-empty terminal subsets or, in other words,

that there is no j, $1 \leq j \leq I$, such that DECODE(j) yields a non-terminal subset. This is done by showing that during the decoding process, if node w is included in the subset and node u is a successor of node w then u must subsequently be included in the subset. The following property of the labelling is required for this proof.

Let node u belong to the Q(u)-th chain.

Lemma 3: If node u is a successor of node w and $Q(u) < Q(v) < Q(w)$, then node u is a successor of node v.

Proof: Refer to procedure LABELTREE, Procedure 2. Since node u is a successor of node w, in the preorder traversal of the tree the call to the procedure to traverse w is made and completed while the call to traverse u is suspended. Also, since $Q(u) < Q(v) < Q(w)$ and the chains are defined in increasing order of their index, the call to traverse node v must have been made after the call to traverse u and before the call to traverse w. Consequently, node v is also visited and the traversal of v completed while the traversal of u is suspended.

Hence, node u is the root of a subtree containing both v and w. Therefore, node u is a successor of node v. ∎

Lemma 3.4: Consider a tree labelled with procedure LABELTREE. During the decoding of an index, j, of a subset, if node u is a successor of node w and w is included in the

subset, then u must subsequently be included in the subset.

Proof: Refer to procedure DECODE, Procedure 3. The proof is by induction on the number of chains. Note that for any i, $1 \leq i \leq N$, the first i chains constitute a terminal subtree and the labels of the subset is a labelling for the subtree.

The theorem is trivial for the first or only chain. Suppose it is true for the tree formed by the first (i-1) chains, $1 < i < N$. During the decoding, if no tasks of chain i are selected, then the theorem applies to the first i chains.

Suppose a task, w, of chain i is selected and u is a successor of w. If $Q(u) = Q(w) = i$, then u is also selected at the same time with w. If, on the other hand, u belongs to a smaller numbered chain (all higher numbered chains have been dealt with at this time), then there are two cases:

(1) There exists some task v with $Q(u) < Q(v) < Q(w)$ such that v is subsequently selected. By Lemma 3, u is also a successor of v and by the induction hypothesis for the first $Q(v)$ chains, u must be subsequently selected.

(2) Now suppose that there is no task v with $Q(u) < Q(v) < Q(w)$ which is included in the subset. Let $Q(u) = j$. Thus, after selecting a number of tasks including w from chain i, no other tasks are selected until chain j,

$j < i$, is under consideration. Let c be the resulting subset index after chains N,N-1,...,i+1 have been considered and let k_i be the number of tasks from chain i which are selected. Then, $c \geq s_i + k_i a_i$. This inequality still holds at the time chain j is being considered. If k_j tasks of chain j (including task u) are successors of node w and hence of the last task on chain i, then $c - k_i a_i \geq s_i = s_j + k_j a_j$. Thus, the condition for selecting at least k_j tasks from chain j is satisfied. Therefore, node u must be selected.

In any case, the theorem now applies to the first i chains. By induction, it applies to all N chains. ∎

Theorem 3.3: For a tree labelled with LABELTREE, the terminal subsets are given by DECODE(j), $1 \leq j \leq I$.

Proof: By Lemma 3.1 every terminal subset has a unique index between 1 and I. By Lemmas 3.2 and 3.4, DECODE(j), $1 \leq j \leq I$, yields a unique terminal subset. Thus, there is a one-to-one correspondence between the first I integers and the non-empty terminal subsets of the given tree. ∎

It is easy to check for Example 3.2 that the number I = 65 obtained is indeed the number of non-empty terminal subsets of the tree. The number of terminal subsets of a tree satisfies the following recurrence relation. Let h(u) be the number of non-empty terminal subsets of the tree rooted at node u. Since a terminal subtree of a tree rooted

at node u consists of u and some terminal subtrees rooted at predecessors of u, $h(u) = 1 + \prod\{h(v)\}$, where v ranges over all immediate predecessors of u. Applying this recurrence relation to the tree of Example 3.2 yields $h(r) = 65$.

The main part of the solution of the tree problem is outlined in Procedure 4. Here, L(g) is the label given to a node, g, and F(j) is the length of an optimal schedule for the subtree whose index is j. The procedure outline follows closely the previous discussion and requires no further comments.

In order to construct the actual schedule another array of I pointers should be maintained, indicating for each index, j, which subtree was used to obtain F(j). With this array of pointers and the DECODE procedure, subsequent construction of an optimal schedule is straightforward.

Now, consider the time and space requirements of TREE-ALG. It is easily seen that both LABELTREE and DECODE each take O(n) time where n is the number of nodes. For any terminal subtree, G, the number of pairs, (g_1,g_2) of processor-1 and processor-2 leaves is no more than n^2. Hence, TREE-ALG requires at most $O(n^2I)$ time.

As for storage, note that since a tree of n nodes has n-1 edges, the storage of a tree structure requires only O(n) storage locations. Similarly, LABELTREE and DECODE also require linear storage. During the terminal subtree

```
    procedure TREE-ALG;
1.  begin comment L(g) is the label of node g.
                  I is the index of the complete tree.
                  F(j) is the length of an optimal schedule for
                  subtree with index j;
2.  LABELTREE; (note I is computed by LABELTREE)
3.  for j := 1 until I do
4.        begin G := DECODE(j);
5.        determine processor-1 leaves of G;
6.        determine processor-2 leaves of G;
7.        F(j) := min{F(j - L(g1) - L(g2))},

                  where the minimum is taken over all pairs of
                  processor-1 leaves, g1, and processor-2
                  leaves, g2, of G;
          end;
    end;
```

PROCEDURE 4

enumeration, $O(n^2)$ locations are needed for the pairs, (g_1,g_2). However, the same locations can be used for every subtree since it is only necessary to save F(j). Hence, storage requirement is at most of $O(n^2 + I)$.

Thus, the complexity of the algorithm is really determined by the number of subsets generated. Although the number of subsets is exponential in n, in general, the presence of precedence constraints significantly reduces the number of terminal subsets. In Example 3.2, only 65 subsets would be generated as compared to $2^9-1 = 511$ non-empty subsets of the nine tasks.

Finally, consider the other types of precedence graphs. Solutions for initially rooted trees may be obtained by initially reversing all the directions, applying the terminally rooted tree method and finally reversing the

resulting schedule. Similarly, solutions for forests may be obtained by adding a dummy root which is connected to the roots of all the trees, applying the corresponding tree algorithm and finally deleting the dummy root from the schedule. As for the general acyclic digraph, it is obvious that a good dynamic programming solution along the lines discussed above hinges on the availability of a good terminal subset enumerator. Recently, Achugbue and Chin (1980), have developed a terminal subset enumerator for arbitrary precedence graphs which takes at most O(eI) time and O(e) space, where e is the number of edges.

3.6 Discussion

In this chapter, the NP-completeness of the minimal length scheduling problem for UET processor-bound systems introduced by Goyal (1977) for the case of two processors and an arbitrary number of chains has been demonstrated. This appears to be the simplest case of the processor-bound systems that is NP-complete and indeed the result subsumes all NP-completeness results of Goyal as well as the case of a fixed number of processors $m \geq 2$ and precedence constraints in the form of trees, which he left as an open problem.

In addition, a dynamic programming approach is proposed for finding minimal length schedules for these systems which represents a significant improvement over simple

enumeration. This approach requires a terminal subset enumeration scheme. In the case of trees (and forests), a terminal subset enumeration algorithm is presented. It is efficient in the sense that it never enumerates a non-terminal subset. For general precedence graphs, the scheme of Baker and Schrage (1978) is good in that it is fast but it however includes some non-terminal subsets in its enumeration. The recent algorithm of Achugbue and Chin (1980) is reasonably fast and never enumerates non-terminal subsets.

Chapter Four

FLOW SHOP SCHEDULES

In the flow shop model, considered in this chapter, and the open shop of the following chapter, several related tasks are grouped together to form a job. Thus, a flow shop consists of m processors, P_j, $1 \leq j \leq m$, and n jobs, J_i, $1 \leq i \leq n$, where job J_i contains m tasks or stages, $T_i[j]$, $1 \leq j \leq m$. The flow shop is processor bound since task $T_i[j]$ must be executed on the j-th processor, and it is characterized by a unidirectional flow of tasks, that is, task $T_i[j]$ must be executed before $T_i[j+1]$ for any i.

When dealing with two- or three-stage shops, it is often more convenient to refer to the component tasks of job J_i as tasks A_i, B_i and C_i (to be executed on processors P_1, P_2 and P_3 respectively).

The flow shop is perhaps the first of the processor bound models to be studied extensively probably due to the fact that it closely simulates assembly line systems. Much of what is known about the model is reported in Conway, Maxwell and Miller's book (1965) and the more recent text by Baker (1974). The emphasis in this chapter is on non-preemptive schedules minimizing schedule length for three-stage systems.

4.1 Survey

Johnson (1954) showed that for two and three-processor flow shop minimal length non-premptive scheduling problems it is sufficient to consider only permutation schedules, in which the jobs are scheduled in the same order on all the processors and a processor is not kept idle if a task for that processor is ready for execution (thus, permutation schedules are keep-busy schedules). He further gave the well-known O(nlogn) solution for the two-processor system. The rule for obtaining an optimal schedule in a two-stage flow shop is to schedule the i-th job before the j-th if $MIN(A_i,B_j) \leq MIN(A_j,B_i)$.

For three-stage flow shops, solutions have been presented for several special cases. These are itemized below by original author:

(1) Johnson (1954) extended his two-stage rule to the case of three processors and showed that the extended version produces minimal length schedules, also in O(nlogn) time whenever the task system satisfies either $A_i \geq B_j$ for all i and j, or $C_i \geq B_j$ for all i and j.

He further conjectured that when his two-stage rule applied to the first two stages yields the same permutation as that for the last two stages, then the permutation is optimal for the three stage problem. However, Burns and Rooker (1976) showed that this is

not always the case. Johnson's conjecture is true if for each application of the two stage rule, the inequality is strict for all job pairs, or if the permutation resulting from the first two applications is also optimal for the first and third processors.

(2) Arthanari and Mukhopadhyay (1971) gave an $O(n^2)$ algorithm for systems with either $A_i \leq B_j$ for all i,j or $C_i \leq B_j$ for all i,j.

(3) Smith, Panwalker and Dudek (1975) considered systems with ordered processing time matrices, in which the relative order (in terms of task length) of the tasks in every job is the same, and in addition, if one task of job J_i is less than the corresponding task of job J_j, then every task of job J_i is less than the corresponding task of job J_j.

(4) The system of Burns and Rooker (1975) must satify the condition that the product of $MIN(A_i,B_j)-MIN(B_i,C_j)$ and $MIN(A_j,B_i)-MIN(B_j,C_i)$ be non-negative. Their algorithm is easily seen to require $O(n^2)$ time at most.

More recntly (1978), they gave an O(nlogn) algorithm for the case in which $B_i \leq MIN(A_i,C_i)$ for all i.

(5) Swarc (1977) also gave an O(nlogn) algorithm for the case in which $B_i = B_j$ and if $A_i \leq A_j$ then $C_i \geq C_j$ for all i and j.

Another case due to Swarc can be described as

follows. Let permutation, p, be optimal for the two stage problem in which job J_i has tasks A_i+B_i, B_i+C_i, with associated optimal length U_p. If

$U_p = L_p + \Sigma_{i=1}^{n}(B_i)$,

then p is optimal for the three stage problem, where L_p is the length of the three stage schedule using permutation p.

For the general problem with m≥2 several heuristic methods have also been devised (see Baker, 1974). However, the problem is NP-complete as shown by Gonzalez, Johnson and Sahni (1976) and Gonzalez and Sahni (1978).

In the following sections, several new results are given for some interesting special three stage flow shops.

4.2 J-Maximal and J-Minimal Flow Shops

A flow shop is said to be <u>j-maximal</u> (<u>j-minimal</u>) if the j-th task of each job is not smaller than (greater than) any other task of the same job. This criterion is far less restrictive than the ordered processing time flow shop of Smith, Panwalker and Dudek (1975). J-maximal and j-minimal flow shops have been studied by Chin and Tsai (1978) and bounds on the performance of the worst solutions as compared to the best possible were derived.

It is easily checked that known proofs of NP-completeness of the minimal length flow shop scheduling

problem (Gonzalez, Johnson & Sahni, 1976; Gonzalez & Sahni, 1978) involve the setting up of flow shops which do not have the j-maximal or j-minimal criterion for any value of j. Hence, their results do not carry over to the cases under consideration.

In the following, it is shown that the three-stage problem remains NP-complete except for the 2-minimal case which is solved in O(nlogn) steps.

Chin and Tsai show that the 2-minimal problem is NP-complete under the assumption that if a job has a zero-length task on a certain processor, then the job does not have to visit that processor. With this interpretation of zero-length tasks it is no longer true that an optimal schedule may be found among the permutation schedules. However, a more realistic interpretation of zero-length tasks for the flow shop model is to consider them as tasks with infinitesimal time requirement. Thus, each job must visit every processor even when a job has a zero-length task for a processor. The latter interpretation is adopted in this thesis.

4.2.1 2-Minimal Flow Shop: A Solvable Case

Given a permutation p of the first n integers, $(p(1),p(2),\ldots,p(n))$, and any u, v with $1 \le u \le v \le n$, the number $L_p(u,v) = \sum_{i=1}^{u} A_{p(i)} + \sum_{i=u}^{v} B_{p(i)} + \sum_{i=v}^{n} C_{p(i)}$ is a lower bound on the length of the schedule derived from p. In fact, the length of that schedule, L_p, is $MAX\{L_p(u,v) \mid 1 \le u \le v \le n\}$. This follows from the fact that job p(u) (that is, the p(u)-th job or $J_{p(u)}$ to be exact) cannot start until after processor 1 has executed stage 1 of all preceding jobs, task $B_{p(v)}$ cannot start until processor 2 finishes all previous stage 2 tasks (in particular, those between p(u) and p(v)), and finally, after the processing of task $C_{p(v)}$, the third processor has to do the remaining third-stage tasks. This lower bound with different u and v will be used repeatedly in the following.

Johnson's (1954) proposal for the three-stage problem is to schedule the i-th job before the j-th job if

$$MIN(A_i + B_i, B_j + C_j) \le MIN(A_j + B_j, B_i + C_i) \quad ---(1)$$

One of the special cases solved by the above rule as recently demonstrated by Burns and Rooker (1978) is (by definition of j-minimal) a 2-minimal flow shop. The proof is straightforward and is sketched below for completeness.

Theorem 4.1 (after Burns & Rooker, 1978): The 2-minimal three-stage flow shop scheduling problem can be solved in O(nlogn) time by the application of Johnson's rule (1).

Proof: Let $p = (p(1),\ldots,p(n))$ be a permutation of the first n integers. Consider the two-processor flow shop with n jobs such that for the i-th job, the task on the first processor takes (A_i+B_i) time and the other task takes (B_i+C_i). Then rule (1) above simply applies Johnson's optimal procedure for two processors. In other words, rule (1) finds a permutation which minimizes $MAX\{L_p(j,j) \mid 1\leq j\leq n\}$. Therefore, in order to prove the theorem it is sufficient to show that

$$L_p(u,v) \leq MAX\{L_p(u,u),\ L_p(v,v)\}, \qquad ---(2)$$

for all $u<v$ and all permutation p.

Suppose (2) does not hold.

Then, $L_p(u,v) > L_p(u,u)$ and $L_p(u,v) > L_p(v,v)$,

for some u, v. This yields

$$\Sigma_{i=u+1}^{v}(B_{p(i)}) > \Sigma_{i=u}^{v-1}(C_{p(i)})$$

and $$\Sigma_{i=u}^{v-1}(B_{p(i)}) > \Sigma_{i=u+1}^{v}(A_{p(i)}),$$

from which one obtains $B_{p(v)} > C_{p(u)}$ and $B_{p(u)} > A_{p(v)}$. Since in a 2-minimal flow shop $A_i \geq B_i$ and $C_i \geq B_i$, $1\leq i\leq n$, it follows that $B_{p(v)} > C_{p(u)} \geq B_{p(u)} > A_{p(v)}$. Hence, $B_{p(v)} > A_{p(v)}$ which is a contradiction.

An O(nlogn) implementation of the given rule is easily envisaged. ∎

4.2.2 NP-Complete Cases

The easy solution of the 2-minimal case for three processors might lead one to conjecture that other j-maximal or j-minimal cases might just as easily be solved. In this section it is shown that this is definitely not the case.

The NP-completeness of the 2-maximal three-stage flow shop problem is easily demonstrated by a reduction from PARTITION (see Section 1.2.2).

Theorem 4.2: The 2-maximal three-stage flow shop minimal length scheduling problem is NP-complete.

Proof: Given an instance of PARTITION, define the following 2-maximal flow shop containing n+1 jobs.

$$A_i = C_i = 0, \text{ for } 1 \leq i \leq n.$$

$$B_i = a_i, \text{ for } 1 \leq i \leq n.$$

$$A_{n+1} = B_{n+1} = C_{n+1} = K.$$

The deadline for the flow shop problem is D = 3K.

By considering Figure 18, it is obvious that there exists a permutation schedule for the flow shop whose length does not exceed 3K if and only if the PARTITION instance has a solution. ∎

Incidentally, the flow shop of Theorem 4.2 is also 1-minimal as well as 3-minimal and serves to show NP-completeness of these cases. They were not included in the statement of that theorem because stronger reductions

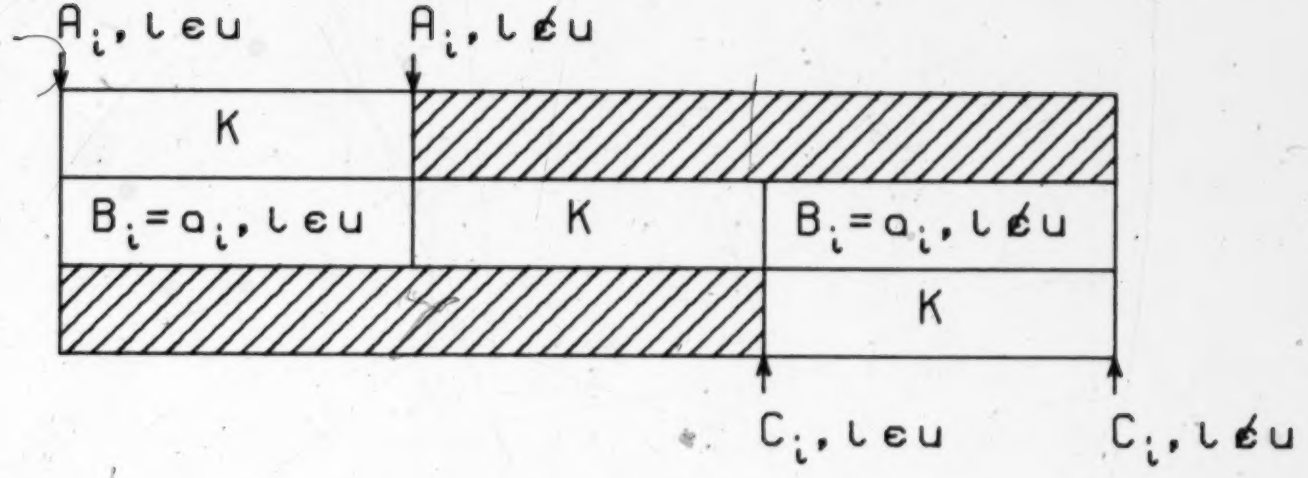


FIGURE 18: Optimal schedule of Theorem 4.2

from 3-PARTITION will be presented.

Lemma 4.1: Consider the following 1-minimal (n+2)-job flow shop:

$$A_i = 2(i - 1)K, \quad 1 \leq i \leq n+1;$$

$$B_i = (2i - 1)K, \quad 1 \leq i \leq n+1;$$

$$C_i = 2(i + 1)K, \quad 1 \leq i \leq n+1;$$

$$A_{n+2} = C_{n+2} = 0$$

$$B_{n+2} = C_{n+1} = 2(n + 2)K; \text{ for } K > 0.$$

The permutation $(1,\ldots, n+2)$ is the unique optimal permutation.

Proof: See Figure 19 for the case n = 4. Note that the given schedule has total idle time of length K on the third processor, and that for any schedule the third processor must be at least idle during the execution of the first second-stage task. Since K is the minimum second-stage task the given permutation must be optimal.

Now, consider an arbitrary optimal permutation $(p(1),\ldots,p(n+2))$. If $p(1) \neq 1$ then $B_{p(1)} > K$ and there is idle period greater than K on the third processor, contradicting optimality of p. Hence, p(1) must be 1. Similarly, if $p(2) \neq 2$ then $B_{p(2)} > C_{p(1)} = C_1$ and extra idle period is generated on the third processor. Hence, p(2) must be 2. Continuing in this manner, it is clear that the permutation p must be precisely the one given in the lemma.

∎

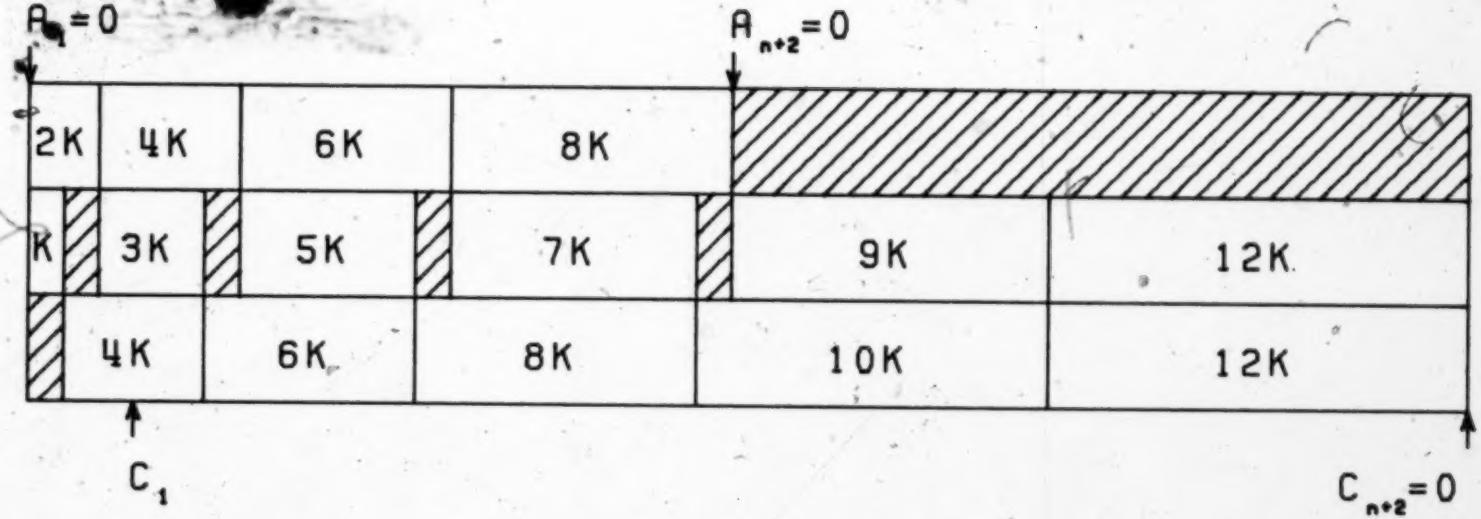


FIGURE 19: Unique optimal permutation schedule of Lemma 4.1, n=4

Note that the length of the optimal schedule is $(n^2 + 5n + 6)K$ and that any other permutation yields a schedule strictly longer than this.

Theorem 4.3: The 1-minimal flow shop minimal length scheduling problem is NP-complete.

Proof: Given an instance of 3-PARTITION with the set $\{a_1,\ldots,a_{3n}\}$, the a_i summing to nK and $K/4 < a_i < K/2$, construct the following instance of a 1-minimal flow shop minimal length scheduling problem.

Use all the jobs of Lemma 4.1 and include the following: $A_{n+2+j} = C_{n+2+j} = \emptyset$, $B_{n+2+j} = a_j$, for $1 \le j \le 3n$. The target schedule length is $D = (n^2 + 5n + 6)K$, the length of the optimal schedule of Lemma 4.1.

The unique optimal schedule of Lemma 4.1 leaves n idle periods each of length exactly K into which the 3-element partitions (stage 2 of the new jobs) can be placed if the 3-PARTITION problem has a solution. Thus, there exists a schedule whose length is D.

Conversely, by the uniqueness property in Lemma 4.1, the only type of schedules for the flow shop which can possibly finish by time D are those which contain the jobs of Lemma 4.1 in their optimal order as given in Figure 19. That order can therefore be considered as a skeleton of any optimal schedule. Consider the possible placements of the

new jobs in the skeleton. It is impossible to insert the new jobs at either end without lengthening the schedule. Furthermore, the first set of n+1 jobs leave n periods on processor 2 into which the remaining can be placed only if set $\{a_i\}$ has a 3-partition (note that the $K/4 < a_i < K/2$ property of the a_i is essential here). ∎

Theorem 4.4: The 3-minimal flow shop minimal length scheduling problem is NP-complete.

Proof: Similar to Lemma 4.1 and Theorem 4.3. For the lemma part use the n+2 jobs:

$$A_i = 2(i + 1)K,\ 1 \le i \le n+1;$$

$$B_i = (2i - 1)K,\ 1 \le i \le n+1;$$

$$C_i = 2(i - 1)K,\ 1 \le i \le n+1;$$

$$A_{n+2} = C_{n+2} = 0,\ B_{n+2} = A_{n+1} = 2(n + 2)K.$$

The unique optimal permutation is (n+2, n+1, ..., 1). The same set of jobs as for Theorem 4.3 is added to the above set of n+2 jobs. ∎

Lemma 4.2: Consider the following 1-maximal (n+1)-job flow shop:

$$A_i = (i^2 + 3i + 4)K/2,$$

$$B_i = (i^2 + i + 4)K/2,$$

$$C_i = (i^2 - i + 2)K/2, \quad \text{for } 1 \le i \le n+1,\ K>0.$$

The permutation (n+1, n, ..., 2, 1) is the unique optimal permutation.

Proof: See Figure 20 for a schedule with the tasks in

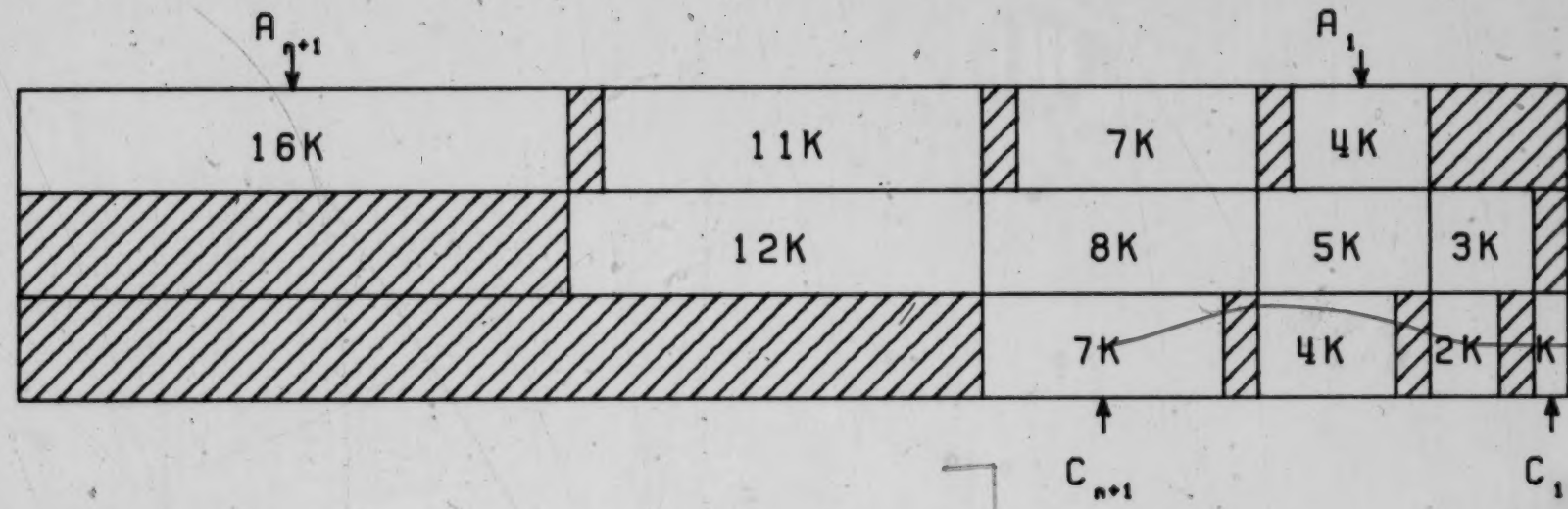


FIGURE 20: Unique Optimal permutation schedule of Lemma 4.2, n=3

the specified order for n=3. The optimality of the permutation given in Figure 20 can be derived from the fact that this permutation is optimal on the first two processors, (i.e. considering the 2-processor flow shop with each job consisting of the first two tasks) and the fact that the optimal schedule length on the three processors is no less than the optimal schedule length on the first two processors plus the smallest task length on the third processor. Since the length of the schedule in Figure 20 is the sum of the optimal schedule for the first two processors and the smallest task length on the third, the given permutation is optimal.

Notice that the length of the given permutation schedule is

$$L_{opt} = A_{n+1} + \Sigma_{i=1}^{n+1}(B_i) + C_1 \qquad ---(3)$$
$$= (n^3 + 9n^2 + 38n + 48)K/6.$$

It remains to prove the uniqueness property. Consider the given optimal permutation p. p(1) must be n+1 for otherwise the length of the corresponding schedule, L_p, is bounded as follows:

$$L_p \geq L_p(u,n+1) = \Sigma_{i=1}^{u} A_{p(i)} + \Sigma_{i=u}^{n+1} B_{p(i)} + C_{p(n+1)}$$

where p(u) = n+1, u>1. But $C_{p(n+1)} \geq C_1$ and $A_{p(i)} > B_{p(i)}$, for any i. Hence, by comparing with the expression for L_{opt} in Equation (3), $L_p > L_{opt}$, a contradiction. Thus, p(1) = n+1. Similarly, it can be shown that p(n+1) = 1. Thus, the permutation p starts with n+1 and finishes with 1.

The rest of the proof consists of showing by induction that n must precede n-1 and then that n-1 must precede n-2 and so on. Incidentally, the fact that $p(1) = n+1$ implies that n+1 must precede n. Assume that j+1 precedes j for $u+1 \leq j < n+1$. Let $p(s) = u+1$. A lower bound on the length of schedule p is

$$L_p(1,s) = A_{n+1} + \Sigma_{i=1}^{s} B_{p(i)} + \Sigma_{i=s}^{n+1} C_{p(i)} \qquad ---(4)$$

Now, $C_{p(n+1)} = C_1$ and the summation of the $B_{p(i)}$ in Equation (4) includes the second stage of all jobs j for $j \geq u+1$ by hypothesis. If in addition u precedes u+1 then $\Sigma_{i=s}^{n+1} C_{p(i)}$ is a sum of C_{u+1}, C_1, and a subset of $\{C_2, \ldots, C_{u-1}\}$.

For $1 \leq i \leq n+1$,

$$B_i - C_i = (i^2 + i + 4)K/2 - (i^2 - i + 2)K/2$$
$$= (i + 1)K.$$

Therefore, $\Sigma_{i=1}^{u-1}(B_i - C_i) = (u(u - 1)/2 + u - 1)K$

$$= (u^2 + u)K/2 - K.$$

Hence,
$$\Sigma_{i=1}^{u-1} B_i = (u^2 + u)K/2 - K + \Sigma_{i=1}^{u} C_i$$
$$= (u^2 + u)K/2 + \Sigma_{i=2}^{u-1} C_i \quad (\text{note } C_1 = K)$$
$$< (u^2 + u + 2)K/2 + \Sigma_{i=2}^{u-1} C_i$$
$$= C_{u+1} + \Sigma_{i=2}^{u-1} C_i.$$

Notice further that the inequality holds if B_j is removed from the left and C_j from the right for $1 < j < u$ since $B_j \geq C_j$. Thus, the sum of B_1 and any subset of $\{B_2, \ldots, B_{u-1}\}$ is strictly less than the sum of C_{u+1} and the corresponding subset of $\{C_2, \ldots, C_{u-1}\}$.

Comparing the expressions for $L_p(1,s)$ and L_{opt}, it follows that $L_p(1,s) > L_{opt}$. Hence, either u+1 precedes u or

a contradiction is obtained. By induction this property holds for all u. ∎

Theorem 4.5: The 1-maximal flow shop minimal length scheduling problem is NP-complete.

Proof: Given an instance of 3-PARTITION with set $\{a_1, \ldots, a_{3n}\}$, the a_i summing to nK and $K/4 < a_i < K/2$, construct the following instance of a 1-maximal flow shop problem.

Use all the jobs in Lemma 4.2 and include the following.

$A_{n+1+j} = C_{n+1+j} = a_j$, $B_{n+1+j} = 0$, for $1 \leq j \leq 3n$.
The target schedule length is $D = (n^3 + 9n^2 + 38n + 48)K/6$, which is the optimal schedule length of Lemma 4.2.

After scheduling the first n+1 jobs as specified in Lemma 4.2, n equal periods of length K can be obtained on processor 1 by shifting the first stage tasks as far right as possible (this has been done in Figure 20) and there are n similar periods appropriately positioned on the third processor. Thus, the final 3n jobs can be scheduled in these spaces if the 3-PARTITION problem has a solution. Observe that the resulting schedule is indeed a permutation schedule.

Conversely, it is only necessary to consider schedules for the flow shop which contain the first n+1 jobs in their

optimal order for otherwise the deadline, D, will definitely be exceeded, by the uniqueness property of Lemma 4.2. It is obvious that none of the final 3n jobs can be inserted into the skeleton before the first job (i.e. n+1) or after the last job (job 1). Furthermore, if some of the final 3n jobs with total processing time at the first stage exceeding K are inserted between any two jobs of the skeleton, additional idle periods will be generated on the second processor (see Figure 20) and the schedule becomes suboptimal. Hence, a schedule of length D can be obtained only if the 3-PARTITION problem has a solution. ∎

Theorem 4.6: The 3-maximal flow shop minimal length scheduling problem is NP-complete.

Proof: Similar to Lemma 4.2 and Theorem 4.5. For the lemma part use the jobs,

$$A_i = (i^2 - i + 2)K/2,$$
$$B_i = (i^2 + i + 4)K/2,$$
$$C_i = (i^2 + 3i + 4)K/2,\ 1 \leq i \leq n+1.$$

The unique optimal permutation is $(1, 2, \ldots, n+1)$. The same set of jobs as for Theorem 4.5 is added to the above set of n+1 jobs. ∎

4.3 Ordered Three Stage Flow Shops

The results of the previous section show that, with but one exception, the minimal length scheduling problem for j-maximal and j-minimal three-stage flow shops is NP-complete. In this section, the flow shop is further restricted so that it is j-maximal and at the same time k-minimal for some $j \neq k$, $1 \leq j \leq 3$, $1 \leq k \leq 3$. Thus, for each job the task on a certain processor is the largest and the task on one of the remaining two processors is the smallest. This leads to an ordering of the tasks A_i, B_i, C_i for each job i. Note that the resulting shop is still less restrictive than the case considered by Smith, Panwalker and Dudek (1975).

Let L stand for the processor with the largest task of each job, S the processor with the smallest task, and M (for medium) for the remaining processor. Then flow shops of the type described above can be classified as follows:

processor	123
	LMS = 1-maximal & 3-minimal
	LSM = 1-maximal & 2-minimal
type of	MLS = 2-maximal & 3-minimal
flow shop	MSL = 3-maximal & 2-minimal
	SLM = 2-maximal & 1-minimal
	SML = 3-maximal & 1-minimal

It is clear that the O(nlogn) algorithm for scheduling the 2-minimal flow shop works for LSM and MSL shops. Similarly, the proof of NP-completeness for the 2-maximal flow shop applies to MLS and SLM shops. However, the LMS and SML flow shops present a new problem. In the following sections, $O(n^6)$ algorithms are given for scheduling these types of flow shop. First, consider LMS flow shops.

4.3.1 LMS Flow Shops

In computing the expression L_p, the length of permutation p (i.e. the length of the schedule derived from permutation p), the following convention is adopted. If $L_p(x,y) = L_p(r,s)$ and $s>y$ then, use $L_p(r,s)$ when the actual indices that yield L_p are of particular interest.

A set of n numbers $\{B_i \mid 1 \le i \le n\}$ is said to be <u>almost sorted</u> in descending order if for any B_i, $1 \le i < n$, there exists at most one value of j, $i < j \le n$, such that $B_i < B_j$.

It will be shown that there exists an optimal permutation p with $L_p = L_p(u,v)$ which satisfies the following conditions:

(1) $B_{p(i)} \le B_{p(i+1)} \Rightarrow C_{p(i)} \ge C_{p(i+1)}$, $1 \le i < n$. In particular $B_{p(i)} < B_{p(i+1)} \Rightarrow C_{p(i)} > C_{p(i+1)}$, $1 \le i < n$.

(2) $C_{p(v)} > B_{p(k)}$, $v < k \le n$.

(3) $B_{p(i)} \ge C_{p(v)}$, $1 \le i \le v$.

(4) $\{B_{p(i)} \mid 1 \le i \le n\}$ is almost sorted in descending order.

(5) If m is the minimum index for which $B_{p(m)} < B_{p(u)}$ and $m<u$, then $B_{p(m)},\ldots,B_{p(u-1)}$ is sorted in descending order. In addition, $B_{p(i)} \leq B_{p(j)}$, $i>u$, $m \leq j<u$.

The algorithm constructs an optimal permutation, p, which satisfies all of the above conditions.

Condition (1) indicates that in the scheduling process, jobs with the same execution time on the second stage may be arranged in descending order of the third stage.

Suppose the job which will be in the v-th position in p is known (i.e. in computing $L_p(u,v)$ it is known in advance which job will contribute the second index v). Then, the jobs to precede p(v) and those to follow job p(v) can be determined by conditions (2) and (3). Thus, the remaining n-1 jobs are partitioned into two subsets

(a) those jobs with $B_i \geq C_{p(v)}$, and

(b) those with $B_i < C_{p(v)}$.

The jobs in subset (a) must be executed before job p(v) while those in subset (b) must be executed after job p(v). The actual value of the index v will then be determined by the size of the subsets.

Now, suppose further it is known in advance which job will play the role of p(u). The jobs in subset (a) can then be partitioned further as follows:

(c) those jobs with $B_i \geq B_{p(u)}$, and

(d) those with $B_i < B_{p(u)}$.

Now the jobs in subset (c) must precede job p(u). (Otherwise, if there is i, $u<i\leq v$, with $B_{p(i)} > B_{p(u)}$, then $L_p(i,v) > L_p(u,v)$ and $L_p \neq L_p(u,v)$.) By conditions (4) and (5), if k jobs in subset (a) precede job p(u) they must be the k largest in (a) and must immediately precede p(u) sorted in descending order, and $k = u-m$.

It is clear that the above conditions are not sufficient to determine an optimal permutation. There are still three unknown parameters, p(v), p(u) and the number, k, of the previous paragraph.

Therefore, the algorithm tries all possible combinations of the following choices:

(I) choose job p(v); use task $C_{p(v)}$ to partition the remaining jobs into subsets (a) and (b) as above.

(II) from subset (a) choose job p(u); use $B_{p(u)}$ to partition the remaining jobs in (a) into (c) and (d).

(III) choose $k \geq 0$ so that the largest k tasks of subset (d) will precede job p(u).

The decisions I, II, III will be said to be optimal whenever there exists an optimal almost sorted permutation for which the choices are correct. It is shown later how to obtain a permutation p, if one exists, for any set of decisions I, II, and III. The algorithm obtains all such permutations and selects the shortest one.

First, the above conditions are shown to hold for some optimal permutation. Conditions (1) and (2) are proved in the following lemma.

Lemma 4.3: There exists an optimal permutation p for an LMS flow shop such that if $B_{p(i)} \leq B_{p(i+1)}$ then $C_{p(i)} \geq C_{p(i+1)}$.

Proof: The proof is by construction. Suppose p is optimal, $B_{p(i)} \leq B_{p(i+1)}$, and $C_{p(i)} < C_{p(i+1)}$. Consider permutation q obtained from p by interchanging p(i) and p(i+1) in p. We show that $L_q \leq L_p$ by showing that for all x, y, $1 \leq x \leq y \leq n$, $L_q(x,y) \leq L_p(u,v)$ for some u, v, $1 \leq u \leq v \leq n$. The following cases exhaust the possible values of x and y.

CASE 1: $x \leq y < i$. $L_q(x,y) = L_p(x,y)$.

CASE 2: $x < i$, $y = i$.

$L_q(x,i) = L_p(x,i+1) + C_{p(i)} - B_{p(i)}$.

Since $C_{p(i)} \leq B_{p(i)}$, $L_q(x,i) \leq L_p(x,i+1)$.

CASE 3: $x < i$, $y = i + 1$.

$L_q(x,i+1) = L_p(x,i+1) + C_{p(i)} - C_{p(i+1)}$.

Since $C_{p(i)} < C_{p(i+1)}$, $L_q(x,i+1) < L_p(x,i+1)$.

CASE 4: $x < i$, $y > i + 1$. $L_q(x,y) = L_p(x,y)$.

CASE 5: $x = i$, $y = i$.

$L_q(i,i) = L_p(i+1,i+1) + C_{p(i)} - A_{p(i)}$.

Since $C_{p(i)} \leq A_{p(i)}$, $L_q(i,i) \leq L_p(i+1,i+1)$.

CASE 6: $x = i$, $y = i + 1$.

$$L_q(i,i+1) = L_p(i+1,i+1) + B_{p(i)} + C_{p(i)} - A_{p(i)} - C_{p(i+1)}.$$

Since $C_{p(i)} < C_{p(i+1)}$ and $B_{p(i)} \leq A_{p(i)}$,
$L_q(i,i+1) < L_p(i+1,i+1)$.

CASE 7: $x = i$, $y > i + 1$.

$L_q(i,y) = L_p(i+1,y) + B_{p(i)} - A_{p(i)}$.

Since $B_{p(i)} \leq A_{p(i)}$, $L_q(i,y) \leq L_p(i+1,y)$.

CASE 8: $x = i + 1$, $y = i + 1$.

$L_q(i+1,i+1) = L_p(i+1,i+1) + B_{p(i)} + C_{p(i)} - B_{p(i+1)} - C_{p(i+1)}$.

Since $C_{p(i)} < C_{p(i+1)}$ and $B_{p(i)} \leq B_{p(i+1)}$,
$L_q(i+1,i+1) < L_p(i+1,i+1)$.

CASE 9: $x = i + 1$, $y > i + 1$.

$L_q(i+1,y) = L_p(i+1,y) + B_{p(i)} - B_{p(i+1)}$.

Since $B_{p(i)} \leq B_{p(i+1)}$, $L_q(i+1,y) \leq L_p(i+1,y)$.

CASE 10: $i + 1 < x \leq y$. $L_q(x,y) = L_p(x,y)$.

Therefore, $L_q \leq L_p$ and q must be optimal for otherwise the optimality of p is contradicted. After a finite number of such exchanges, an optimal permutation which satisfies the lemma is obtained. ∎

Corollary: There exists an optimal permutation p for an LMS flow shop such that if $B_{p(i)} < B_{p(i+1)}$ then $C_{p(i)} > C_{p(i+1)}$.

Proof: Same as the proof of the lemma using the inequalities $B_{p(i)} < B_{p(i+1)}$, and $C_{p(i)} \leq C_{p(i+1)}$ instead of $B_{p(i)} \leq B_{p(i+1)}$ and $C_{p(i)} < C_{p(i+1)}$. ∎

Lemma 4.4: Let p be any permutation for an LMS flow shop with length $L_p(u,v)$. Then, $C_{p(v)} > B_{p(k)}$, for $v<k\leq n$.

Proof: Note that in an LMS flow shop $A_i \geq B_i \geq C_i$, $1\leq i\leq n$. Since length of p is $L_p(u,v)$, $L_p(u,k) < L_p(u,v)$, for $v<k\leq n$. This implies

$$\Sigma_{i=1}^{u} A_{p(i)} + \Sigma_{i=u}^{k} B_{p(i)} + \Sigma_{i=k}^{n} C_{p(i)}$$
$$< \Sigma_{i=1}^{u} A_{p(i)} + \Sigma_{i=u}^{v} B_{p(i)} + \Sigma_{i=v}^{n} C_{p(i)}.$$
$$\Rightarrow \Sigma_{i=v+1}^{k} B_{p(i)} < \Sigma_{i=v}^{k-1} C_{p(i)}$$
$$\Rightarrow \Sigma_{i=v+1}^{k-1} B_{p(i)} + B_{p(k)} < \Sigma_{i=v+1}^{k-1} C_{p(i)} + C_{p(v)}.$$

Since $B_{p(i)} \geq C_{p(i)}$ for any i, it follows that $B_{p(k)} < C_{p(v)}$. ∎

The following lemma is needed to facilitate later proofs.

Lemma 4.5: There exists an optimal permutation p for an LMS flow shop such that if $B_{p(i)} < B_{p(i+1)}$ then $C_{p(i)} > B_{p(j)}$, $1\leq i\leq n-2$, $i+1<j\leq n$.

Proof: The proof is again by construction. Let p be optimal with $B_{p(i)} < B_{p(i+1)}$ and such that the permutation q obtained by interchanging p(i) and p(i+1) in p is not optimal. We show that $L_q = L_q(r,i+1)$ for some r, $1\leq r\leq i+1$, by proving that $L_q(x,y) \leq L_p$ for $y\neq i+1$. By the corollary to Lemma 4.3, $C_{p(i)} > C_{p(i+1)}$. The following cases (similar to Lemma 4.3) are obtained.

CASE 1: $x \leq y < i$. $L_q(x,y) = L_p(x,y)$.

CASE 2: $x < i$, $y = i$.

$L_q(x,i) = L_p(x,i+1) + C_{p(i)} - B_{p(i)}$.

Since $C_{p(i)} \leq B_{p(i)}$, $L_q(x,i) \leq L_p(x,i+1)$.

CASE 3: $x < i$, $y = i + 1$. ** unknown **

CASE 4: $x < i$, $y > i + 1$. $L_q(x,y) = L_p(x,y)$.

CASE 5: $x = i$, $y = i$.

$L_q(i,i) = L_p(i+1,i+1) + C_{p(i)} - A_{p(i)}$.

Since $C_{p(i)} \leq A_{p(i)}$, $L_q(i,i) \leq L_p(i+1,i+1)$.

CASE 6: $x = i$, $y = i + 1$. ** unknown **

CASE 7: $x = i$, $y > i + 1$.

$L_q(i,y) = L_p(i+1,y) + B_{p(i)} - A_{p(i)}$.

Since $B_{p(i)} \leq A_{p(i)}$, $L_q(i,y) \leq L_p(i+1,y)$.

CASE 8: $x = i + 1$, $y = i + 1$. ** unknown **

CASE 9: $x = i + 1$, $y > i + 1$.

$L_q(i+1,y) = L_p(i+1,y) + B_{p(i)} - B_{p(i+1)}$.

Since $B_{p(i)} < B_{p(i+1)}$, $L_q(i+1,y) < L_p(i+1,y)$.

CASE 10: $i + 1 < x \leq y$. $L_q(x,y) = L_p(x,y)$.

Since q is not optimal, $L_q > L_p$ but from the above cases it is clear that $L_q(x,y) \leq L_p$ whenever $y \neq i+1$. Hence, there exists r, $1 \leq r \leq i+1$, such that $L_q(r,i+1) > L_p$. It follows tnat $L_q(r,i+1) > L_q(r,j)$, $i+1 < j \leq n$. By Lemma 4.4, $C_{q(i+1)} > B_{q(j)}$, or $C_{p(i)} > B_{p(j)}$, $i+1 < j \leq n$. ∎

It is convenient to first prove condition (4) in the next lemma and use the result in proving condition (3).

Lemma 4.6: There exists an optimal permutation p for an

LMS flow shop such that $\{B_{p(i)} \mid 1 \le i \le n\}$ is almost sorted in descending order.

Proof: The proof is by induction on the size of the set $\{B_{p(i)},\ldots,B_{p(n)}\}$.

Initial case: $\{B_{p(n)}\}$ is almost sorted in descending order.

Induction step: Suppose $\{B_{p(i+1)},\ldots,B_{p(n)}\}$ is almost sorted in descending order. If $B_{p(i)} \ge B_{p(i+1)}$, then the almost sorted condition holds for positions i,...,n. If, on the other hand, $B_{p(i)} < B_{p(i+1)}$, then, by Lemma 4.5, $C_{p(i)} > B_{p(j)}$, $j>i+1$. Now, since $B_{p(i)} \ge C_{p(i)}$, it follows that $B_{p(i)} > B_{p(j)}$, $j>i+1$.

Hence, in any case, $\{B_{p(i)},\ldots,B_{p(n)}\}$ is almost sorted in descending order. ∎

Lemma 4.7 proves a stronger result than condition (3) which is stated in the corollary.

Lemma 4.7: There exists an optimal permutation p for an LMS flow shop such that $B_{p(i)} \ge C_{p(j)}$ for $1 \le i < j \le n$.

Proof: Let p be an optimal permutation satisfying Lemmas 4.3 to 4.6. We first show that $B_{p(j-1)} \ge C_{p(j)}$, $1<j\le n$. If $B_{p(j-1)} \ge B_{p(j)}$, then $B_{p(j-1)} \ge C_{p(j)}$, since $B_{p(j)} \ge C_{p(j)}$. If $B_{p(j-1)} < B_{p(j)}$, then by the corollary to Lemma 4.3 $C_{p(j-1)} > C_{p(j)}$, and since $B_{p(j-1)} \ge C_{p(j-1)}$, we

obtain $B_{p(j-1)} > C_{p(j)}$. Thus, in either case, $B_{p(j-1)} \geq C_{p(j)}$.

Now consider the case of arbitrary indices i and j, $1 \leq i < (j-1) < j \leq n$. This case is trivial if $B_{p(i)} \geq B_{p(j)}$. As for $B_{p(i)} < B_{p(j)}$, since $\{B_{p(i)} \mid 1 \leq i \leq n\}$ is almost sorted in descending order, $B_{p(i)} \geq B_{p(k)}$, $i < k \leq n$, $k \neq j$. In particular, $B_{p(i)} \geq B_{p(j-1)}$, and $B_{p(j-1)} \geq C_{p(j)}$, (by proof of first part). Hence, $B_{p(i)} \geq C_{p(j)}$. ∎

Corollary: If $L_p = L_p(u,v)$ then $B_{p(i)} \geq C_{p(v)}$, $1 \leq i \leq v$.

Proof: Follows directly from the lemma. ∎

The final condition (5), is given in the next lemma.

Lemma 4.8: Let p be an optimal permutation for an LMS flow shop which satisfies Lemmas 4.3 to 4.7. Let the length of p be $L_p(u,v)$. If m is the minimum index for which $B_{p(m)} < B_{p(u)}$ and $m < u$, then the subsequence $B_{p(m)}, \ldots, B_{p(u-1)}$ is sorted in descending order. In addition, $B_{p(i)} \leq B_{p(j)}$, $i > u$, $m \leq j < u$.

Proof: This follows directly from the almost sorted condition. ∎

This concludes the proofs of the five conditions. We now show how to obtain a permutation p which corresponds to a set of decisions I, II and III, if such a permutation exists.

Recall that after making the decisions I, II and III, the following situation results.

(1) The first (m - 1) jobs are those with $B_i \geq B_{p(u)}$.

(2) The next k = u - m jobs consist of the k jobs which have largest second stage among those with second stage less than $B_{p(u)}$.

(3) Next job is p(u).

(4) Job p(u) is followed by the remaining jobs with $B_{p(u)} > B_i \geq B_{p(v)}$.

(5) Next comes job p(v).

(6) The remaining jobs follow job p(v).

The second group may be empty depending on decision III. The actual values of the indices m, u and v are determined by the sizes of the above sets. The first step is to sort each group in descending order of B_i, using descending order of C_i (Lemma 4.3) whenever there is a tie between the B_i. This results in an initial almost sorted permutation p which is further modified by the following procedure ALSORT where necessary.

In order to clearly explain the operation of procedure ALSORT, two operations, cycle and sort, are defined on a pair of indices i, j as follows. Applying the operation cycle(i,j) to a sequence $\{B_1,\ldots,B_i,\ldots,B_j,\ldots,B_n\}$ produces the sequence $\{B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_j,B_i,\ldots,B_n\}$. Applying the operation sort(i,j) to the above sequence reverses the effect of the previous cycle operation. In other words,

cycle rotates the elements in positions i through j one place to the left and "sort" rotates them one place in the opposite direction. Intuitively, cycle operations will be performed on the permutation initially obtained by sorting the jobs as indicated above so as to reduce the length of the schedule.

The almost sorted condition implies that there exists a disjoint set of index pairs $\{(s(i),t(i)) \mid 1 \le i \le I\}$, where $s(i-1) < t(i-1) < s(i) < t(i)$, such that p may be obtained from the permutation supplied to ALSORT by applying cycle(s(i),t(i)), $1 \le i \le I$. The number of pairs, I, is initially unknown but bounded by n/2. It will be shown that for an optimal set of decisions I,II,III procedure ALSORT will find an optimal index set and perform the required cycle operations.

For any permutation p let E(i,j) be the earliest time that task i of job p(j) can be finished and Y(j) the sum of execution times of all third stage tasks following $C_{p(j)}$, in the permutation schedule corresponding to p. A procedure to calculate and update the values of E(i,j) and Y(j) is needed. Set $E(i,0) = E(0,j) = 0$, $0 \le i \le 3$, $0 \le j \le n$, and set $Y(n) = 0$. Then, the values of E(i,j) and Y(j) can be computed with the relations:

$Y(j) = Y(j+1) + C_{p(j+1)}$, $0 \le j < n$, and

$E(i,j) = MAX\{E(i-1,j),E(i,j-1)\} + t$, $1 \le i \le 3$, $1 \le j \le n$,

where t is $A_{p(j)}$, $B_{p(j)}$ or $C_{p(j)}$ depending on i. This

computation is provided by procedure LENGTH(p,x,y) (Procedure 5) which calculates B(i,j), $1 \le i \le 3$, $x \le j \le y$, and Y(j), $x \le j \le y$, with the assumption that the boundary (i=0, $j<x$, $j>y$) contains appropriate values. Clearly, this procedure requires only O(y-x+1) time. Note that $B(3,n) = L_p$ and $B(3,j) + Y(j) = MAX\{L_p(i,j) \mid 1 \le i \le j\}$.

Once the decision set I, II and III has been specified, $L_p(u,v)$ can be calculated. Thus, ALSORT starts by computing $L_p(u,v)$. Essentially, the algorithm seeks an almost sorted permutation, p, with $L_p = L_p(u,v)$. This is done by ensuring that $MAX\{L_p(r,j) \mid 1 \le r \le j\} \le L_p(u,v)$, $1 \le j \le n$. Thus, for each position j starting from 1 up to n it checks if

$$MAX\{L_p(r,j) \mid 1 \le r \le j\} > L_p(u,v), \qquad ---(5)$$

or $B(3,j) + Y(j) > L_p(u,v)$.

There are two cases to be considered depending on the position j, $1 \le j \le n$.

Case 1: $m \le j \le u$ (or j=u, when $u<m$) or j=v. For this case, if the Inequality (5) holds then the decision set I, II, III cannot be optimal i.e. there does not exist an almost sorted sequence p with $L_p = L_p(u,v)$. The procedure CHECK(p,x,y) (Procedure 6) is used to check that the Inequality (5) does not hold between any two given positions x and y. The logical value returned by the procedure indicates whether or not the condition has been detected in the given range. This procedure also takes O(y-x+1) time.

```
   Procedure LENGTH(p,x,y);
1. begin comment A_i,B_i,C_i, 1≤i≤n and E(i,j), 0≤i≤3, 0≤j≤n
        and Y(j), 1≤j≤n, are global variables.
        A_i,B_i,C_i are task lengths for the LMS flow shop;
        E(i,j) is earliest time task i of job j can
        finish. E(i,0) and E(0,j), 0≤i≤3, 0≤j≤n, are
        initialized to 0.
        Y(j) is the sum of task lengths C_p(k), j<k≤n.
        Y(n) is initialized to 0.
        p is the permutation under consideration.
        E(i,j) and Y(j) are updated for x≤j≤y.
2. for j := x until y do
3.     begin for i := 1 until 3 do
4.         begin t := (A_p(j),B_p(j),C_p(j)) depending on i;
5.         E(i,j) := maximum{E(i-1,j),E(i,j-1)} + t;
6.         end;
7.     end;
8. for j := y-1 step -1 until x-1 do
       Y(j) := Y(j+1) + C_p(j+1);
9. end;
```

PROCEDURE 5

Case 2: $1 \le j < m$ (or $1 \le j < u$ if $u < m$), $u < j < v$ or $v < j \le n$. For these positions, if the Inequality (5) holds at j, it is still possible that there exists an almost sorted permutation with $L_p = L_p(u,v)$. To discover such a permutation, an attempt is made to perform a cycle(i,j) operation, where $1 \le i < j < m$ (or $1 \le i < j < u$) or $u < i < j < v$ or $v < i < j \le n$, depending on which of above ranges j falls in, so that the inequality no longer holds for all positions from 1 to j inclusive. This may require that a previous cycle operation be reversed (a sort(i,j) operation) in order to accommodate a cycle operation at a higher index since the permutation must at all times remain almost sorted. These checks and

```
    procedure CHECK(p,x,y);
 1. begin comment procedure checks to ensure
          L_p(r,j) ≤ L_p(u,v) x≤j≤y, 1≤r≤j.

          Logical value FLAG indicates success or failure,
          LEN is L_p(u,v).
          Other variables are as in Procedure 5.

 2. FLAG := true;
 3. LENGTH(p,x,y);
 4. j := x;
 5. while FLAG and j ≤ y do
 6.      begin if E(3,j) + Y(j) > LEN then FLAG := false;
 7.      j := j + 1;
 8.      end;
 9. return FLAG;
10. end;
```

PROCEDURE 6

necessary cycle operations are carried out with the procedure CHECKCYCLE(p,x,y) (Procedure 7). Note that after a cycle(i,j) operation, this routine uses the CHECK procedure to test if (5) no longer holds. It is easily seen that the procedure takes no more than $O(n^2)$ time.

Thus, procedure ALSORT (Procedure 8) computes $L_p(u,v)$ and with the aid of the procedures discussed above, attempts to find a permutation p with $L_p = L_p(u,v)$ by

(1) checking that Inequality (5) is not true for $m \le j \le u$ (or $j=u$ for $u<m$) and $j=v$, and

(2) ensuring (performing appropriate cycle operations when necessary) that Inequality (5) does not hold for $1 \le j < m$ (or $1 \le j < u$ for $u<m$), $u<j<v$, and $v<j \le n$.

In order to prove that procedure ALSORT works as required, the following lemma which is in a sense a stronger

```
    procedure CHECKCYCLE(p,x,y);
 1. begin comment procedure performs the same checks as
          procedure CHECK but tries to perform CYCLE
          operations at problem spots;
          A stack of cycled index pairs is maintained in
          arrays S-ST(*) and T-ST(*) with the top pair being
          S-ST(TOP) and T-ST(TOP);
 2. FLAG := true;
    comment initialize stacks;
 3. TOP := 0; S-ST(TOP) := 0; T-ST(TOP) := 0;
    comment check next position and if necessary cycle;
 4. for j := x until y do
 5.       begin LENGTH(p,j,j);
 6.       if B(3,j) + Y(j) > LEN then
 7.            begin comment there exists r such that
                       L_p(r,j) > L_p(u,v);
 8.            FLAG := false; i := j - 1;
 9.            while not(FLAG) and i ≥ x do
10.                 begin if i = T-ST(TOP) then
11.                      begin SORT(S-ST(TOP),T-ST(TOP));
12.                      LENGTH(p,S-ST(TOP),T-ST(TOP));
13.                      comment pop stack; TOP := TOP - 1;
14.                      end;
15.                 CYCLE(i,j); LENGTH(p,i,j);
16.                 if CHECK(p,1,j) then
17.                      begin comment push stack;
18.                      TOP:=TOP+1;
19.                      S-ST(TOP):=i; T-ST(TOP):=j;
20.                      FLAG := true;
21.                      end else
22.                      begin SORT(i,j); LENGTH(p,i,j);
23.                      i:=i-1;
24.                      end;
25.                 end;
26.            end;
27.       end;
28. return FLAG;
29. end;
```

PROCEDURE 7

```
   Procedure ALSORT(p,m,u,v);
1. begin comment global variables are explained in
         Procedure 5.
         p, m, u, and v have the same significance as in
         the text. LEN is the length of the modified
         permutation at exit;

   comment first compute L_p(u,v);

2. LEN := Σ_{i=1}^{u} A_{p(i)} + Σ_{i=u}^{v} B_{p(i)} + Σ_{i=v}^{n} C_{p(i)};

   comment check and if necessary cycle in positions 1
         to m-1;
3. HIGH := u-1; if m < u then HIGH := m-1;
4. if HIGH ≥ 0 then
5.     begin LOW := 1;
6.     if not(CHECKCYCLE(p,LOW,HIGH)) then goto NOGOOD;
7.     end;
   comment check positions m to u (or just u);
8. LOW := m; if m > u then LOW := u;
9. if not(CHECK(p,LOW,u)) then goto NOGOOD;
   comment check (and cycle) in positions u+1 to v-1;
10. if v-u > 1 then
11.     if not(CHECKCYCLE(p,u+1,v-1)) then goto NOGOOD;
   comment check position v;
12. if not(CHECK(p,v,v)) then goto NOGOOD;
   comment check (and cycle) in positions v+1 to n;
13. if v < n then
14.     if not(CHECKCYCLE(p,v+1,n)) then goto NOGOOD;
   comment permutation found;
15. LEN := E(3,n); goto DONE;
   comment failed to obtain permutation, abort;
16. NOGOOD: LENGTH(p,1,n); LEN := E(3,n);
17. DONE: end;
```

PROCEDURE 8

version of Lemma 4.5 is used.

Lemma 4.9: Let p be an almost sorted optimal permutation for an LMS flow shop with the positions s,...,t cycled. Let q be a permutation such that $q(i) = p(i)$, $1 \le i < s$, jobs $p(s),...,p(t)$ are rearranged as $p(s),..,p(k-1),p(t),p(k),..,p(t-1)$ in positions s,...,t of q and the remaining jobs are in any order in q. Then, the following hold.

(1) $L_q(x,i) < L_p$, $1 \le i \le k$, $1 \le x \le i$.

(2) There exists i, $k < i \le t$, such that $L_q(x,i) > L_p$ for some x, $1 \le x \le i$.

Proof: First assume that jobs $p(t+1),...,p(n)$ are in the same order in q as in p.

Note that by previous lemmas (the corollary to Lemma 4.3 in particular), p does not contain any unnecessary cycled positions. Hence, q must be sub-optimal since in q (s,k) rather than (s,t) is cycled. Now, the only positions in q which differ from those of p are in positions k to t, thus:

p: $B_{p(k)},.....,B_{p(t-1)},B_{p(t)}$

q: $B_{p(t)},B_{p(k)},.....,B_{p(t-1)}$

If there is any position $i < k$ such that $L_q(x,i) > L_p$, then the change from q to p does not affect the value of $L_q(x,i)$ and thus $L_q(x,i) = L_p(x,i) > L_p$, a contradiction. Similarly, suppose $L_q(x,k) > L_p$ for some x. Since $B_{p(t)} > B_{p(k)}$, $B_{p(k)} \ge C_{p(k)}$, and $B_{p(i)} \ge C_{p(i)}$, $k < i < t$, it follows that $L_q(x,k) \le L_p(x,t)$. Therefore, $L_p < L_q(x,k) \le L_p(x,t)$, again

a contradiction. Hence, part (1) follows.

Now since q is not optimal there exists some $i>k$ such that $L_q(x,i) > L_p$ for some x, $1 \le x \le i$. But suppose that there is no such $i \le t$. Since the only difference between q and p occur in position $(k,\ldots,t)$, x must satisfy $k \le x \le t$, for otherwise $L_p(x,i) = L_q(x,i)$. Now, for $k \le x \le t$ we have $L_q(x,i) \le L_p(t,i)$ since $B_{p(i)} \le A_{p(i)}$, $x \le i < t$, and if $x>k$, $B_{q(x)} = B_{p(x-1)} < B_{p(t)}$, and $B_{p(t)} \le A_{p(t)}$. But $L_q(x,i) \le L_p(t,i)$ implies that $L_p(t,i) > L_p$, a contradiction. Hence, there exists i, $k<i \le t$, such that $L_q(x,i) > L_p$, for some x, $1 \le x \le i$. Thus part (2) is proved.

It is clear that if the jobs in positions $t+1,\ldots,n$ in q are not in the same sequence as in p, the above conclusions still hold since the order of these jobs does not affect the values $L_q(x,i)$ for $i \le t$. ∎

Lemma 4.10: Procedure ALSORT will perform an optimal set of cycle operations when the set of decisions, I, II, III, is optimal. The procedure requires at most $O(n^3)$ time.

Proof: Suppose an optimal set of decisions I, II, III has been made. Then there exists an optimal almost sorted permutation, p, with $L_p = L_p(u,v)$ where p(u) and p(v) are the jobs chosen in II and I and the jobs $B_{p(i)}$, $m \le i < u$, are the jobs chosen in III, which are less than $B_{p(u)}$. Now, p differs from the permutation passed to ALSORT only in that the cycle operation has been performed once on a disjoint

set of index pairs {(s(i),t(i)), | 1≤i≤I} as previously noted. It is sufficient to show that ALSORT will find all the required positions s(i),t(i) and perform the correct cycle operations, thereby deriving p. The case of only one cycle is straightforward. Suppose the first k-1 cycles have been found, yielding an intermediate permutation q, then by Lemma 4.9, there will be a position, i, in the range of the k-th cycle for which $L_q(x,i) > L_p$ = LEN, for some x, 1≤x≤i. Since this is precisely the condition that ALSORT tests, the procedure will seek for the index s(k) and subsequently cycle(s(k),i) in order to remove the condition.

Notice that even if the true cycle is on (s(k),t(k)) while the algorithm initially tries to cycle(s(k),i), s(k)<i<t(k), then part (1) of Lemma 4.9 ensures that a temporary solution consisting of cycle(s(k),i) exists and part (2) ensures that the procedure will subsequently seek to cycle(s(i),t(i)) since any partial cycling does not entirely remove the condition from the positions s(k),...,t(k).

The complexity of ALSORT is dominated by the calls to the CHECK and CHECKCYCLE procedures. Since only one of these is called for each j, 1≤j≤n, and since CHECK is linear and CHECKCYCLE requires at most $O(n^2)$ time, it follows that procedure ALSORT takes at most $O(n^3)$ time. ∎

The main algorithm is presented in Procedure 9.

```
   Algorithm LMS-FLOW;
1. begin comment p is the permutation under test, OPTPERM
        is the optimal permutation after running the
        algorithm. LEN and OPTLEN are the lengths of p and
        OPTPERM respectively. PSAVE is used to save the
        permutation p before decision III is made so that
        it can be restored for alternative choices. Other
        variables are explained in Procedure 5. It is
        assumed that the initial pre-sorting in descending
        order of C_i within B_i has been done;
2. for i := 0 until 3 do E(i,0) := 0; Y(n) := 0;
3. for j := 0 until n do E(0,j) := 0;
   comment initialize permutation to sorted order;
4. for i := 1 until n do p(i) := i;
   comment obtain an upper bound for schedule length;
5. LENGTH(p,1,n); OPTLEN := E(3,n); OPTPERM := p;
   comment try all combinations of decisions I, II, III;
6. for i := 1 until n do
7.     begin comment index i makes decision I;
8.     find v ≥ i such that B_p(x) ≥ C_p(v) > B_p(y),
                1≤x≤v<y≤n);
9.     CYCLE(i,v); LENGTH(p,i,v);
10.    for k := 1 until n do psave(k) := p(k);
11.    for j := 1 until v do
12.        begin comment index j makes decision II;
13.        for u := j until v do
14.            begin comment index u makes decision III;
15.            if j < v then
16.                begin if u < v then
17.                    begin CYCLE(j,u); LENGTH(p,j,u);
18.                    m := j; if m = u then m := m+1;
19.                    end else goto NEXT;
20.                end else m := v+1;
21.            ALSORT(p,m,u,v);
22.            if LEN < OPTLEN then
23.              begin OPTLEN := LEN;
24.                for k:=1 until n do OPTPERM(k):=p(k);
25.                end;
               comment restore p for next choice III;
26.            for k := 1 until n do p(k) := psave(k);
27.            LENGTH(p,1,n);
28.            end;
29.        NEXT: end;
       comment restore sorted input permutation;
30.    for k := 1 until n do p(k) := k;
31.    end;
32. end;
```

Theorem 4.7: Algorithm LMS-FLOW finds an optimal permutation for an LMS flow shop in time at most $O(n^6)$.

Proof: The algorithm tries all combinations of choices I, II and III, and must at some stage have an optimal decision set. But by Lemma 4.10 procedure ALSORT produces an optimal permutation whenever the decision set is optimal. Hence, this algorithm will find at least one optimal permutation.

The three choices produce at most n^3 initial permutations to be modified by procedure ALSORT which takes only $O(n^3)$ time. Hence, an optimal permutation will be found in $O(n^6)$ time. ∎

Note that the algorithm trivially requires only $O(n)$ space.

4.3.2 SML Flow Shops

Algorithm LMS-FLOW can be used to schedule an SML flow shop as follows. Given an SML flow shop with jobs $(\underline{A}_i, \underline{B}_i, \underline{C}_i)$, $1 \le i \le n$, with $\underline{A}_i \le \underline{B}_i \le \underline{C}_i$, apply algorithm LMS-FLOW to the flow shop with $A_i = \underline{C}_{n+1-i}$, $B_i = \underline{B}_{n+1-i}$, $C_i = \underline{A}_{n+1-i}$, $1 \le i \le n$, to get an optimal permutation $p = (p(1), \ldots, p(n))$. An optimal permutation for the SML flow shop is $\underline{p} = (p(n), p(n-1), \ldots p(1))$.

Theorem 4.8: The above application of algorithm

LMS-FLOW produces an optimal schedule for an SML flow shop in time $O(n^6)$.

Proof: An SML problem can be regarded as one of minimizing the length of a schedule built from the end backwards with each job executing first on processor 3, and then on processor 2 and finally on processor 1. From this point of view, an SML flow shop problem becomes an LMS flow shop problem and this is precisely the approach taken.

The preconversion before application of algorithm LMS-FLOW as well as the final conversion of p to $\underline{p}$ are linear. Hence, algorithm an optimal is found in at most $O(n^6)$ time. ∎

4.4 Discussion

The foregoing results on some special structure three-stage flow shops are summarized in Figure 21. In the Venn diagram, each circle represents a j-minimal or j-maximal flow shop ($1 \leq j \leq 3$). Their intersections leading to the LMS and the other types of flow shop considered and the associated complexities are indicated. Although the algorithm for the LMS (and SML) flow shops is polynomial in n, its running time of $O(n^6)$ makes it highly impractical. Improving the given algorithm or devising a faster one possibly with a completely different approach remains an open problem.

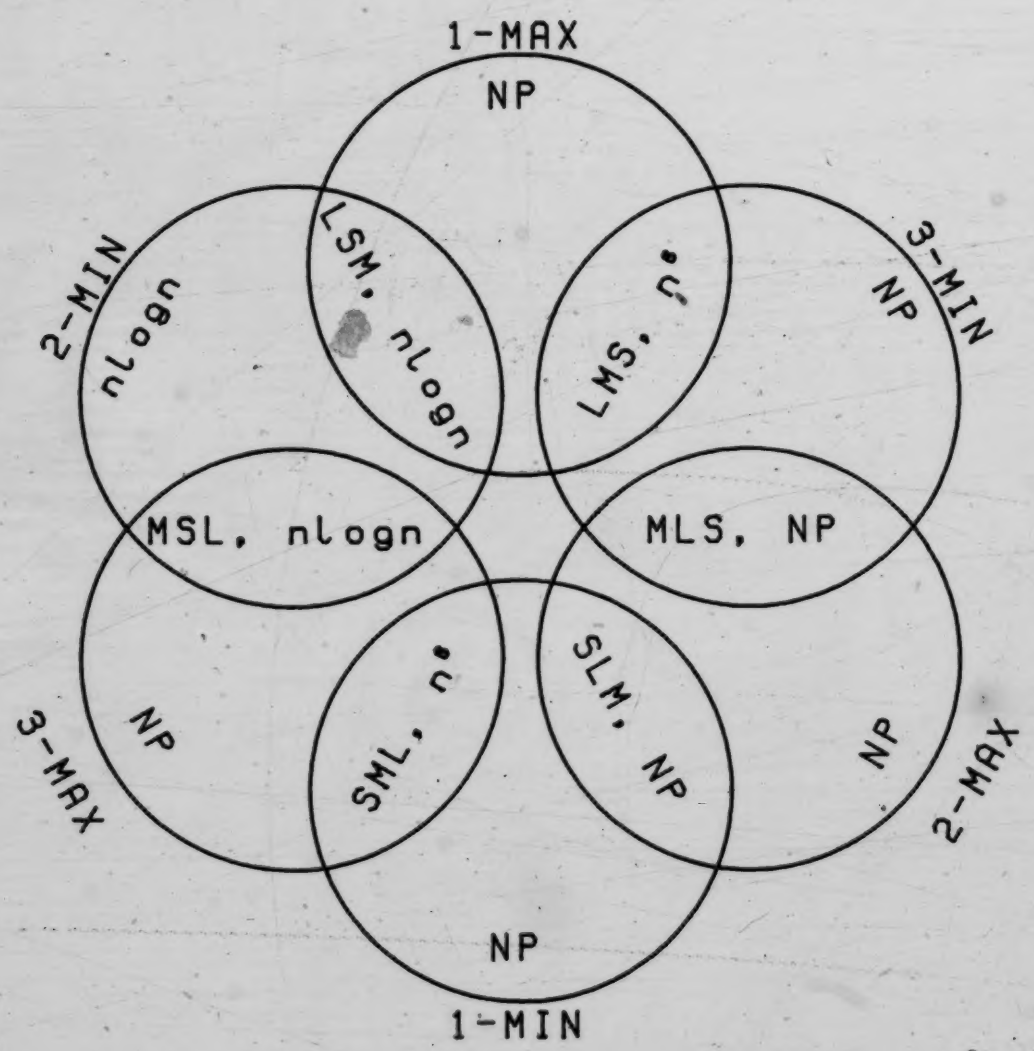


FIGURE 21:
Relationships and complexities
of the special 3-stage flow shops .

Chapter Five

OPEN SHOP SCHEDULES

Another processor bound system of interest is the open shop which is closely related to the flow shop. The only difference between them is that in an open shop, no retrictions are placed on the order in which the tasks of any job are to be done. Thus, an open shop consists of $m \geq 1$ processors, P_i, $1 \leq i \leq m$, and $n \geq 1$ jobs where each job has m component tasks and the j-th task of the i-th job is to be performed on the j-th processor for any i. This chapter deals mainly with the problem of finding optimal non-preemptive schedules for the flow shop with the mean flow time performance measure.

5.1 Survey

The problem of finding minimal length schedules for the open shop has been studied by Gonzalez and Sahni (1976) who provide some motivation for interest in this particular model and Gonzalez (1976). Gonzalez and Sahni presented a linear algorithm for the 2-processor preemptive and non-preemptive systems, and for $m \geq 3$ gave an efficient algorithm for preemptive schedules, while the non-preemptive problem was shown to be NP-complete. Gonzalez subsequently presented a faster algorithm for the preemptive case when

$m \geq 3$.

As for the mean flow time minimization problem, Gonzalez (1979) showed NP-completeness for the case of an arbitrary number of processors. For the case of only one processor, a well-known solution, reported by Smith (1956) is to schedule the tasks in order of non-decreasing execution time.

In the following, the mean flow time problem is shown to be NP-complete for two processors. Subsequently, in Section 5.3, tight bounds on the mean flow time of an arbitrary schedule and an SPT schedule are obtained in terms of the optimal mean flow time.

5.2 Complexity of Mean Flow Schedules

The 2-processor mean flow problem is shown NP-complete by a reduction from 3-PARTITION (Section 1.2.2). The open shop to be constructed consists of a large number of jobs but their number and the sum of their lengths will be polynomial in nK. There are four types of jobs, the T, U, X and Y-jobs. U-jobs are further divided into V and W-jobs. The X and Y-type jobs are large jobs which must be the last jobs in the schedule in order to keep the mean flow small. Furthermore, their execution must not be delayed beyond certain specific times if the bound D is not to be exceeded. The T-jobs, which have very large tasks on processor 1

compared to the T and U-tasks on processor 2, are used to generate fixed-size gaps on the second processor which can only be filled by the U-jobs if and only if the given instance of 3-PARTITION has a solution.

For convenience in specifying the different types of jobs, a job will be given as an ordered pair (x,y) where x is the time taken by the job on processor 1 and y is the time taken on processor 2. The notation S(X), F(X) introduced in Chapter 1 for the starting and finishing time of a task (or job), X, will be adhered to.

Theorem 5.1: The 2-processor open shop non-preemptive mean flow scheduling problem is NP-complete.

Proof: Given a 3-PARTITION problem with n, K and the set $\{a_1 \ldots a_{3n}\}$, consider the following 2-processor flow shop:

T-jobs: $T_0 = (0, g)$; $T_i = (t, g)$, $1 \le i \le n$;

X-jobs: $X_i = (x, 0)$, $1 \le i \le h$;

Y-jobs: $Y_i = (0, y)$, $1 \le i \le h$;

U-jobs: $V_{i,j} = (0, v)$, $1 \le i \le n$, $1 \le j \le (u-3)$

and $W_i = (0, v+a_i)$, $1 \le i \le 3n$;

where $u = 3nK + 3n + 1$, $g = 3nK + 3n + 6$,

$v = n(K + 1)g$, $t = uv + g + K$,

$x = 2(n + h)t$, $y = x$,

$s = (n + 1)g + 3nK + nug + n(n - 1)u(g + K)/2$,

and $h = s + 1$.

The bound D is given as $D = X_e + Y_e + T_e + U_e$, where

$X_e = \Sigma_{i=1}^{h}(nt + ix)$;

$Y_e = \Sigma_{i=1}^{h}(nt + g + iy)$;

$T_e = \Sigma_{i=0}^{n}(g + it)$;

and $U_e = 3nK + \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{u}(g + it + jv))$.

Suppose the 3-PARTITION problem has a solution. Use the schedule suggested in Figure 22. Schedule, on the first processor, tasks $T_i[1]$, $1 \le i \le n$, followed by $X_i[1]$, $1 \le i \le h$, and on the second processor, tasks $T_i[2]$, $0 \le i \le n$, followed by $Y_i[2]$, $1 \le i \le h$. This yields the template of Figure 22. In each of the n hatched areas on the second processor place (u-3) V-type tasks followed by the three W-tasks corresponding to the three elements of one of the 3-element partitions. Clearly, the X, Y and T-jobs contribute X_e, Y_e and T_e to the

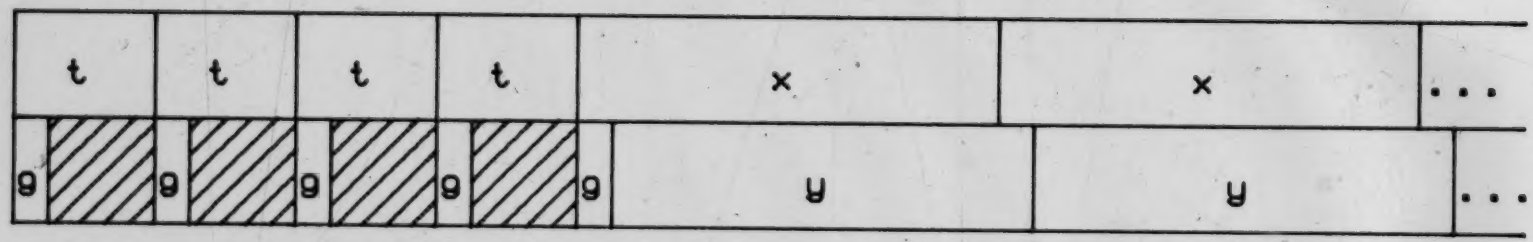


FIGURE 22:
Skeleton of optimal schedule of Theorem 5.1, n=4

mean flow time. Since there are exactly three W-type tasks with sum (3v+K) in each hatched area, the contribution of the i-th hatched area, $0 \le i \le (n-1)$, is less than $(3K + \Sigma_{j=1}^{u}(g + it + jv))$. (Note that the three W-tasks will be the last to be executed in each hatched area.) Hence the U-jobs' contribution to the mean flow time is less than U_e, and the bound D is not exceeded by the mean flow time of the schedule.

Now, suppose there exists a schedule with mean flow time not exceeding the bound D. For brevity such a schedule will be referred to as a good schedule. The proof of the theorem is completed by showing that the desired 3-element partitions must exist. This, in turn, is done by showing that if there is a good schedule, then it can be reduced in polynomial time (i.e. polynomial in nK) to a good schedule with the structure of Figure 22. The following intermediate results (lemmas) are required for this purpose.

5.1 In a good schedule, $S(X_i[1]) \ge nt$ and $S(Y_i[2]) \ge (nt + g)$, $1 \le i \le h$. Hence, all T[1]-tasks are executed before any X[1]-tasks and all T[2] and U[2]-tasks are executed before any Y[2]-task. Without loss of generality, it may be assumed that the X[1], Y[2] and T-tasks are executed in increasing order of their index i.

5.2 In a good schedule, $S(X_1[1]) = nt$ and $S(Y_1[2]) = nt + g$.

In addition, $F(T_i[1]) = ti$, $0 \le i \le n$.

5.3 In a good schedule, during the first $(nt + g)$ time units, at least u tasks are executed on processor 2 in any time interval of length $(t - g)$.

5.4 A good schedule can be reduced in polynomial time to one in which $F(T_i[1]) \le S(T_i[2])$, $0 \le i \le n$, and $F(T_i[2]) \le F(T_{i+1}[1])$, $0 \le i < n$.

5.5 A good schedule can be reduced in polynomial time to one in which $S(T_i[2]) - F(T_i[1]) < v + K/2$, $0 \le i \le n$.

5.6 In a good schedule satisfying all previous lemmas, at most $(u + 1)$ and at least $(u - 1)$ U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \le i < n$.

5.7 In a good schedule satisfying all previous lemmas, EXACTLY u U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \le i < n$.

5.8 In a good schedule satisfying all previous lemmas, $S(T_i[2]) = ti$, $0 \le i \le n$.

Lemmas 5.1 and 5.2 establish the fact that the T[1]-tasks are done in the first nt time units and are immediately followed by the X[1]-tasks. In addition, the Y[2]-tasks are the final tasks to be done on processor 2 and they must start precisely at time $(nt + g)$, thus leaving no idle time on that processor.

The main conclusions of Lemmas 5.3 to 5.8 are given in 5.7 and 5.8 (the others are required merely as intermediate steps in the proofs of these). By Lemma 5.8, $S(T_i[2]) = ti$,

$0 \le i \le n$. At this point, one obtains a good schedule which is similar in structure to that of Figure 22. Consider a good schedule which satisfies the lemmas. Then 3-elements partitions, H_i, $1 \le i \le n$, are given by

$$H_i = \{a_k \mid S(T_{i-1}[2]) < S(W_k[2]) < S(T_i[2])\}.$$

By Lemma 5.7, there are u U-tasks, $U_i[2]$, with $S(T_{i-1}[2]) < S(U_i[2]) < S(T_i[2])$. Since the V[2]-type tasks have length v and the W[2]-tasks have length less than $(v + K/2)$ and greater than $(v + K/4)$ and these u tasks cover an interval of $t - g = uv + K$ without leaving an idle period, the u tasks must contain exactly three W[2]-tasks with total length $3v + K$.

Hence, the open shop has a schedule with mean flow time not exceeding the deadline D if and only if the 3-PARTITION problem has a solution.

The proofs of the lemmas follow. The effects of the lemmas are taken to be cumulative. In other words, when proving any lemma it is assumed that a good schedule which satisfies the previously proven lemmas is available.

Lemma 5.1: In a good schedule, $S(X_i[1]) \ge nt$, and $S(Y_i[2]) \ge (nt+g)$, $1 \le i \le h$. Hence, all T[1]-tasks are executed before any X[1]-task, and all T[2] and U[2]-tasks are executed before any Y[2]-task. Without loss of generality, one may assume that the X[1], Y[2] and T-tasks are executed in increasing order of their index i.

Proof: It can be assumed that all zero-length tasks are executed at time 0. First, suppose that only the first inequality is violated, i.e. $S(Y_i[2]) \geq (nt + g)$ for all i but $S(X_j[1]) < nt$ for some j. Suppose further that only one X-task, $X_j[1]$, starts before time nt. Now, compute a lower bound for the mean flow of the schedule. The Y-jobs contribute at least Y_e. Since $x = 2(n + h)t$ is greater than $(nt + g)$ and no Y-task is executing on processor 2 before time $(nt + g)$, all other processor 2 tasks can be finished before $F(X_j[1])$. Hence, the remaining tasks on processor 1 after time $F(X_j[1])$ are completely independent of tasks remaining on processor 2 and are therefore scheduled optimally in non-decreasing order of their execution time. Hence, the X-jobs contribute at least

$$X_b = x + \Sigma_{i=1}^{h-1}(x + nt + ix) = X_e - nt.$$

Now, since $S(X_j[1]) < nt$, at least one of the tasks $T_i[1]$ must execute after task $X_j[1]$. Hence the T-jobs (consider only the T[1]-tasks) contribute at least

$$T_b = x + \Sigma_{i=1}^{n}(ti) = T_e + x - (n + 1)g.$$

Finally, the contribution of the U-jobs to total mean flow time may be bounded as follows. Since each U-task on the second processor is at least v in length, a trivial lower bound for their flow is

$$\begin{aligned} U_b &= \Sigma_{i=1}^{nu}(iv) = \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{u}(iuv + jv)) \\ &= U_e - 3nK + \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{u}(iuv + jv - g - it - jv)) \\ &= U_e - 3nK + \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{u}(-g - ig - iK)) \end{aligned}$$

$= U_e - nK - nug - n(n-1)u(g+K)/2$

$= U_e - s + (n+1)g.$

Adding these values together gives a lower bound of

$Y_e + X_b + T_b + U_b$

$= Y_e + X_e + T_e + U_e + x - nt - (n+1)g - s + (n+1)g$

$= D + x - nt - s.$

Since x is strictly greater than (nt + s) the schedule cannot be a good one.

Now suppose that only the second inequality in the lemma is violated. Assume $S(Y_i[2]) < nt + g$ for some i. By similar reasoning, one obtains a lower bound of $Y_b = (Y_e - nt - g)$ for the Y-jobs, X_e for the X-jobs, $(T_e - (n+1)g)$ for the T-jobs and $(U_b + y - nuv)$ for the U-jobs, if a U[2]-task executes after $Y_i[2]$. Alternatively, one gets a lower bound of $X_e + Y_b + T_b + U_b$ if a T[2]-task executes after $Y_i[2]$. In either case, y = x is large enough to ensure that the lower bound is strictly larger than D.

Finally, consider the mixed case where some $X_i[1]$ starts before nt and some $Y_k[2]$ starts before (nt + g). It follows that some T[1]-task will execute after $X_i[1]$ and either a T[2]-task or a U[2]-task will execute after $Y_k[2]$. If a U[2]-task follows $Y_k[2]$ then lower bounds of X_b, Y_b, T_b and $(U_b + y - nuv)$ are obtained in the same manner as above. Again, the total lower bound is too large. The only remaining case is when a T[1]-task that follows $X_i[1]$ belongs to the same job as a T[2]-task that follows $Y_k[2]$.

In this case one obtains a lower bound of

$$X_b + Y_b + T_b + U_b = D + x - 2nt - s - g > D.$$

Since the $X_i[1]$, $Y_i[2]$ have the same length it can be assumed that they are executed in order of increasing index i. Similarly, since the T_i-jobs for $1 \leq i \leq n$ have the same length tasks on each processor it can also be assumed that the order of execution on the first processor is in increasing index i.

Lemma 5.2: In a good schedule, $S(X_1[1]) = nt$ and $S(Y_1[2]) = nt + g$. In addition, $F(T_i[1]) = ti$, $0 \leq i \leq n$.

Proof: Note that $X_1[1]$ and $Y_1[2]$ are now the first X and Y-tasks to be executed among the X and Y jobs on the first and second processors respectively.

By Lemma 5.1, $S(X_1[1]) \geq nt$ and $S(Y_1[2]) \geq nt + g$. Suppose $S(X_1[1]) > nt$. Then the flow for the X jobs is at least $\Sigma_{j=1}^{h}(nt + 1 + jx) = X_e + h$. The following trivial lower bounds for the other jobs are easily obtained; Y_e for the Y-jobs, $(T_e - (n + 1)g)$ for the T-jobs (obtained by considering only T[1]-tasks), and U_b for the U-jobs. Adding these together leads to a lower bound of $(D + h - s) = (D + 1)$ for the schedule, which contradicts the fact that it is a good schedule. The case $S(Y_1[2]) > nt + g$ is similar.

Thus, it follows directly that there is no idle period

on the first processor for the first nt time units and on the second processor for the first (nt + g) time units. Thus, $F(T_i[1]) = ti$, $0 \leq i \leq n$. ∎

Lemma 5.3: In a good schedule, during the first (nt + g) time units, at least u tasks are executed on processor 2 in any time interval of length (t - g).

Proof: Recall that there is no idle time on processor 2 during the first (nt + g) time units. The number of tasks executed in the interval is at least $J = (t - g) / (v + K/2)$ since every task executed before (nt + g) on processor 2 has length not exceeding (v + K/2).

Now $$v = n(K + 1)g > nKg > nKu > uK/2$$

$$\Rightarrow \quad K > uK/2 - v - K/2$$

$$\Rightarrow uv + K > (u - 1)(v + K/2)$$

$$\text{or} \quad t - g > (u - 1)(v + K/2)$$

$$\text{or} \quad J > u - 1.$$ ∎

In an open shop, there is no restriction on the order in which a job's tasks are performed. It has been assumed that zero length tasks are executed at time 0. However, the T-jobs have non-zero tasks on both processors. The following lemma shows that there exists a good schedule which executes them in the same processor order, first on processor 1 and then on processor 2.

Lemma 5.4: A good schedule can be reduced in polynomial time to one in which $F(T_i[1]) \leq S(T_i[2])$, $0 \leq i \leq n$ and

$F(T_i[2]) \leq F(T_{i+1}[1])$, $0 \leq i \leq n-1$.

Proof: The proof is by construction. Recall that as a consequence of Lemmas 5.1 and 5.2 there is a good schedule in which $F(T_i[1]) = ti$, $0 \leq i \leq n$.

Consider first the T-jobs whose T[1]-tasks are executed before their T[2]-tasks. Scan the schedule from left to right looking for tasks $T_i[2]$ such that $S(T_i[2]) \geq F(T_i[1]) = ti$. It must be ensured that $F(T_i[2]) \leq t(i+1)$. If this is not the case, perform the following operation as often as necessary.

Operation I:

> Find the first U-task, U_c, on processor 2 preceding $T_i[2]$. Remove U_c from the schedule, shift the tasks following U_c up to and including $T_i[2]$ to the left to take up the interval vacated by U_c, and insert U_c after $T_i[2]$.

It must be shown that it is possible to apply the operation until $F(T_i[2]) \leq t(i+1)$ and that each application does not increase mean flow time. Since $F(T_i[2]) - F(T_i[1]) > t$ and the total length of the T[2]-tasks is $(n+1)g$ which is less than $t - v - K/2$, the existence of a U-task, U_c, with $S(U_c) \geq ti$ and $F(U_c) \leq S(T_i[2])$ is guaranteed. Now, any $T_j[2]$-task between U_c and $T_i[2]$ must have $F(T_j[2]) < S(T_j[1])$, so that shifting such a task to the left does not generate any conflicts with

tasks on processor 1 and does not affect the flow time of job T_j. Now, the flow time of task U_c increases by at most (n+1)g while that of task $T_i[2]$ (and hence of job T_i) decreases by at least v. Since $v = n(K+1)g$ is much greater than (n+1)g, the total mean flow time does not increase.

In the second stage consider those T-jobs whose T[2]-tasks are done before the T[1]-tasks. This time scan the schedule from right to left and perform operation II whenever a $T_i[2]$ with $F(T_i[2]) \leq S(T_i[1])$ is encountered.

Operation II:

Find a V-task, V_c, such that $(ti + g) \leq F(V_c) < (ti + v + g)$. Remove $T_i[2]$ from the schedule, shift the following tasks up to and including V_c to the left to take up the interval vacated by $T_i[2]$, and insert $T_i[2]$ after V_c. If among the shifted tasks a $T_j[2]$ conflicts with $T_j[1]$, exchange the $T_j[2]$ with the following task.

Again, it is necessary to prove the existence of V_c and show that Operation II does not increase mean flow time. Note that any $T_j[2]$ which executes after $T_i[2]$ satisfies the first part of the lemma and must also satisfy the second part (possibly, after an earlier application of II). Consequently, there is no task of length g in the time interval [ti,t(i+1)) (see Figure 23) and at most one task, $T_{i-1}[2]$, of length g in the interval [t(i-1),ti). The task

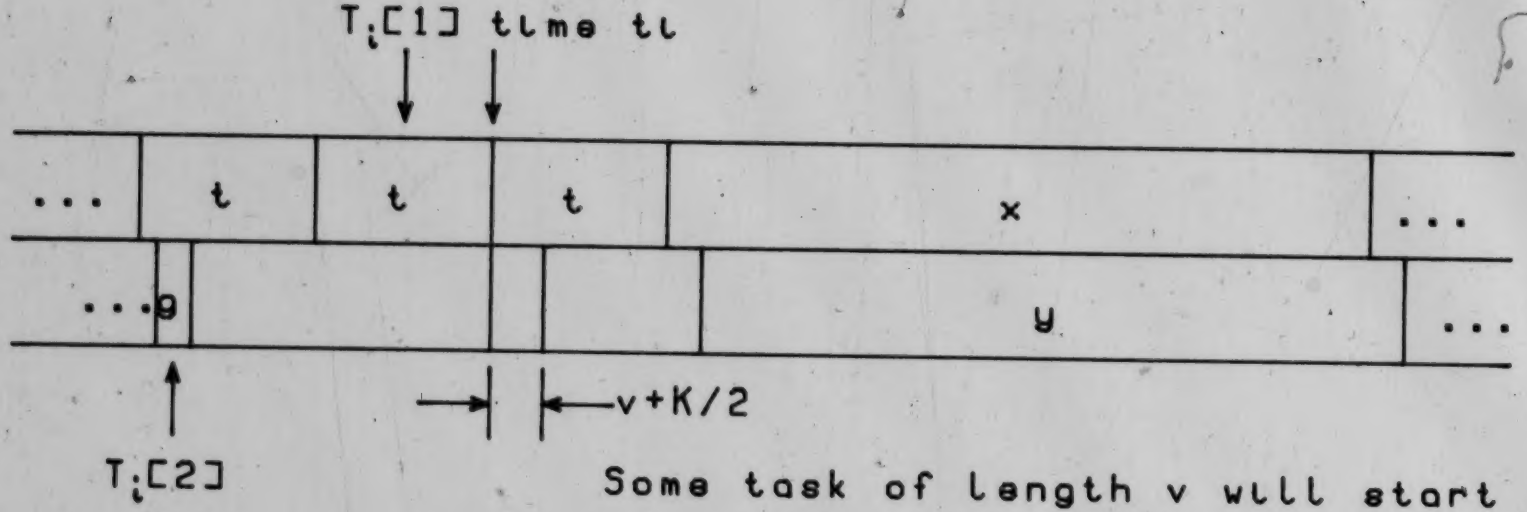


FIGURE 23: Schedule detail, Lemma 5.4

of length g in [t(i-1),ti) must be done before other U-tasks in the same interval. Therefore, there is no task of length g in [ti-v-K/2,t(i+1)) (the lower limit of the interval over which there are no tasks of length g can be decreased but the above is sufficient for the proof). Now the interval [ti-v-K/2,t(i+1)) has length t+v+K/2. Even if all the W-tasks were in this interval, their total length is less than t - 3v and hence there is still room for at least four V-tasks in the interval. Since the V-tasks are smaller than any W-task, they will be the first to be done in the interval. Thus, within [ti-v-K/2, ti+v+K/2) there are only V-tasks or parts of V-tasks, one of which will be identified as task V_c.

Now, consider the first task $V_j[2]$ to finish after time ti. By above arguments this task and the one following it must be V-tasks. $V_j[2]$ finishes before (ti + v + K/2) (which is less than (ti + v + g)). If $F(V_j[2]) - ti \geq g$, then choose $V_c = V_j[2]$, otherwise choose the task following it as V_c. Hence, $F(V_c) - ti < v + g$. In either case, $ti + g \leq F(V_c) < ti + v + g$.

The operation proceeds to remove $T_i[2]$ from the schedule, shift the following tasks, say J in number, up to and including V_c to the left by a distance of g and insert $T_i[2]$ after V_c. Since $F(V_c) \geq ti + g$, after performing II there can be no conflict between $T_i[2]$ and $T_i[1]$. However, it may be necessary to resolve some conflicts between some

$T_j[2]$ and $T_j[1]$ where the $T_j[2]$ is among the tasks shifted left. This is done by exchanging the $T_j[2]$ with the task following it. (It is clear that the following task is a V-task by the same line of argument that was used to determine V_c)

Now, since any intermediate T[2]-tasks already satisfy the second part of the lemma, in any interval $[t(j-1),tj)$, where $F(T_i[2]) \leq t(j-1)$ before II was applied, there can be only one task $T_{j-1}[2]$ of length g. By Lemma 5.3, the same interval must contain at least u U-tasks as well. Hence there are at most $\lfloor J / u \rfloor$ T[2]-tasks among the shifted tasks.

The effects of Operation II on the mean flow time of the schedule can now be computed. The flow time of job T_i increases by at most (v + g). The J tasks shifted to the left lose at least Jg flow time. In addition, the intermediate T[2]-tasks may gain at most $\lfloor J / u \rfloor v$ in exchanges to resolve conflicts. Thus, for the lemma to hold, it must be shown that $Jg \geq v + g + \lfloor J / u \rfloor v$, where $J \geq u$. (Since the J tasks cover at least the interval $[t(i-1),ti]$, J must be no less than u.) Let $J = cu + d$, $c \geq 1$ and $0 \leq d < u$.

$Jg \geq v + g + \lfloor J / u \rfloor v$

$\Leftrightarrow (cu + d)g \geq v + g + cv$

$\Leftrightarrow 3ncK + 3nc + c + d \geq ncK + nc + nK + n + 1, \quad c \geq 1,$

which is easily seen to be true. There are no other changes

to the mean flow time. Hence, a good schedule can be reduced to one satisfying the lemma. ∎

Lemma 5.5: A good schedule can be reduced in polynomial time to one for which $S(T_i[2]) - F(T_i[1]) < v + K/2$, $0 \le i \le n$.

Proof: Consider a good schedule satisfying all previous lemmas. $S(T_i[2]) - F(T_i[1]) \le t - g$. Furthermore, in the interval $(ti, ti + t)$ there is no task of size g on processor 2 other than $T_i[2]$. If $S(T_i[2]) - F(T_i[1]) > v + K/2$, then the task preceding $T_i[2]$, a U[2]-task, can be exchanged with $T_i[2]$ with no increase in mean flow time, and no conflict in the execution of job T_i since that task must start after $ti = F(T_i[1])$. ∎

Lemma 5.6: In a good schedule satisfying all previous lemmas, at most (u + 1) and at least (u - 1) U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \le i < n$.

Proof: The proof is similar to that of Lemma 5.3. By Lemma 5.5, the interval between $T_i[2]$ and $T_{i+1}[2]$ is no smaller than (t-g-v-K/2) and no larger than (t-g+v+K/2).

Suppose there are fewer than (u - 1) U-tasks in the interval. Their total length is at most $nK + (u - 2)v$.

Now, $v = n(K + 1)g > nK - K/2$

$\Rightarrow$ $uv + K/2 - v > nK + uv - 2v$

$\Rightarrow$ $t - g - v - K/2 > nK + (u - 2)v$.

(by definition of t)

Thus, there will be idle time on processor 2 and by Lemma 5.2 the schedule cannot be a good one.

Similarly, suppose there are more than (u + 1) U-tasks. Their total length is at least (u + 2)v.

But $v > 3K/2$

$\Rightarrow$ $uv + 2v > uv + v + 3K/2$

$\Rightarrow$ $(u + 2)v > t - g + v + K/2$. (by definition of t)

Hence, at most (u + 1) U-tasks can be present in the interval. ∎

This condition can be tightened further as follows.

Lemma 5.7: In a good schedule satisfying all previous lemmas, EXACTLY u U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \leq i < n$.

Proof: To begin with, consider in what ways this lemma can be violated. If there are less than iu U-tasks preceding $T_i[2]$ then the total length of tasks preceding $T_i[2]$ (including $T_j[2]$, $j < i$) on processor 2 is at most $(ig + (iu - 1)v + nK)$. This is less than the time interval, ti, which it has to cover as

$$ig + (iu - 1)v + nK < it = i(uv + g + K).$$

Thus, there are at least iu U-tasks preceding $T_i[2]$. Now, let k_i, $0 \leq i \leq (n - 1)$, be the number of U-tasks in the interval between $T_i[2]$ and $T_{i+1}[2]$.

Suppose $k_p = u + 1 = k_q$, and $k_i = u$ for $p < i < q$.

Then, there are ((q + 1 - p)u + 2) U-tasks between $T_p[2]$ and $T_{q+1}[2]$ with total length at least ((q + 1 - p)u + 2)v. Together with the tasks $T_i[2]$, $p \leq i \leq q$, this gives total length at least ((q + 1 - p)u + 2)v + (q + 1 - p)g which is greater than (q + 1 - p)t + v + K/2. Since the latter value expresses the maximum time interval available for the tasks, as implied by Lemma 5.5, this is not possible.

Collecting these facts together, namely, that $k_i = (u - 1)$, u or $(u + 1)$, that $\Sigma_{j=1}^{i}(k_j) \geq iu$, and that it is not possible to have $k_p = k_q = (u + 1)$ and $k_i = u$ for $p < i < q$, one can conclude that the $(u + 1)$ and $(u - 1)$ values of k_i occur alternately starting with a $(u + 1)$ and finishing with a $(u - 1)$, with the u values interspersed.

It is now shown that the bound D is exceeded if k_i is not equal to u for $0 \leq i \leq (n - 1)$. A new lower bound U_c for the flow of the U-jobs can now be computed as

$$U_c = \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{k_i}(g + jv + ti + E_i)).$$

The term $E_i \geq 0$ will compensate for the fact that the $T_i[2]$ cannot start early enough after a $k_p = u + 1$ and before the following $k_q = u - 1$.

$$U_c - U_e = -3nK + \Sigma_{i=0}^{n-1}(\Sigma_{j=1}^{k_i}(g+jv+ti+E_i) - \Sigma_{j=1}^{u}(g+jv+ti)).$$

with respect to the sequence k_i, outside of those subsequences starting with a $(u + 1)$ and ending with a $(u - 1)$, the E_i term is zero and the corresponding summations in the above cancel out. However, it is shown below that if there is at least one

$(u + 1)$, u, ..., u, $(u - 1)$ such subsequence then $U_c + T_c - U_e - T_e > \emptyset$, where T_c is a corresponding bound for the T-jobs.

Let k_p, ..., k_q be one such subsequence. Task $T_i[2]$, $p<i\leq q$, cannot start until after

$$pt + (i - p)g + ((i - p)u + 1)v = ti + v - (i - p)K.$$

Thus $E_p = \emptyset$ and $E_i = v - (i - p)K$, $p<i\leq q$.
In the expression for $(U_c - U_e)$ the summations corresponding to the subsequence are

$$\Sigma_{j=1}^{u+1}(g + jv + pt) - \Sigma_{j=1}^{u}(g + jv + pt)$$
$$+ \Sigma_{i=p+1}^{q-1}(\Sigma_{j=1}^{u}(v - (i - p)K))$$
$$+ \Sigma_{j=1}^{u-1}(g + qt + jv + v - (q - p)K) - \Sigma_{j=1}^{u}(g + qt + jv).$$

After simplification, this becomes

$$-(q - p)(g + uK) - \Sigma_{i=p+1}^{q-1}(u(i - p)K),$$

where the summation in the second term is zero if $q = p + 1$. However, on computing the lower bound T_c for the T-jobs, again the values to be summed outside the subsequences are the same as in the summation for T_e. For the subsequence above, one obtains

$$\Sigma_{i=p+1}^{q}(g + ti + v - (i - p)K) - \Sigma_{i=p+1}^{q}(g + ti)$$
$$= v - (q - p)K + \Sigma_{i=p+1}^{q-1}(v - (i - p)K)$$

as the difference in the two summations. Since $v - (q - p)K > (q - p)(g + uK) + 3nK$ and $v - (i - p)K > u(i - p)K$, $p < i < q$, the overall bound is strictly greater than D. Note that the $3nK$ term in the definition of U_e has been taken care of, while the bounds

for X and Y-tasks remain X_e and Y_e.

Thus if there is at least one subsequence of the type described the flow time is strictly greater than D. Hence, $K_i = u$ for $0 \leq i \leq (n-1)$. ∎

Lemma 5.8: In a good schedule satisfying the previous lemmas, $S(T_i[2]) = ti$, $0 \leq i \leq n$.

Proof: Given a good schedule which satisfies the previous lemmas, assume that $S(T_i[2]) > ti$ for some i, $0 \leq i \leq n$. Then the following lower bounds for the jobs' flow time are easily determined; X_e for the X-jobs, Y_e for the Y-jobs, $(T_e + 1)$ for the T-jobs, and finally at least $(U_e + u - 3nK)$ for the U-jobs (the u term is introduced by the delay on the u U-jobs between $T_i[2]$ and $T_{i+1}[2]$). Adding these together gives a bound of $(D + u + 1 - 3nK)$ which is greater than D. Hence, the lemma must be true. ∎

This concludes the proof of the theorem. ∎

5.3 Heuristic Solutions

The previous section showed the 2-processor n-job mean flow time scheduling problem for the open shop NP-complete. In this section, tight bounds on the mean flow time of an arbitrary schedule and of a schedule obtained using the shortest processing time (SPT) first heuristic as compared to the mean flow time of an optimal schedule are obtained.

Recall that the SPT heuristic is optimal for the 1-processor case.

Let the n jobs be J_i, $1 \le i \le n$. The j-th task of the i-th job is $J_i[j]$ and has execution time $t_i[j]$. Let $T_i = \sum_{j=1}^{m}(t_i[j])$ and $T = \sum_{i=1}^{n}(T_i)$. T_i is the total processing time for the i-th job, while T is the total time for all the n jobs. For any schedule Z the mean flow time of Z will be denoted by mft(Z). The notation for starting and finishing time of a task (or job), a, will be as usual S(a), F(a). (If Z is subscripted, S and F may be given the same subscript to indicate clearly that starting and finishing times are given with respect to the particular schedule Z).

Theorem 5.2: Let Z_o be an optimal mean flow schedule for an m-processor open shop with n jobs. Let Z be an arbitrary schedule. Then, $\frac{mft(Z)}{mft(Z_o)} \le n$.

Proof: Without loss of generality, assume that the jobs are completed in the order $J_1, J_2, \ldots, J_n$. In the worst case, no task of job J_i is started before J_{i-1} has been completed. Hence, $F(J_i) \le \sum_{k=1}^{i}(T_k)$.

Now,
$$mft(Z) = \sum_{i=1}^{n}(F(J_i)) \le \sum_{i=1}^{n}(\sum_{k=1}^{i}(T_k))$$
$$= \sum_{i=1}^{n}(n + 1 - i)T_i \le nT.$$
$$mft(Z_o) = \sum_{i=1}^{n}(F_o(J_i)) \ge \sum_{i=1}^{n}(T_i) = T$$
$$\ge mft(Z)/n. \qquad \blacksquare$$

The bound given above is asymptotically tight as illustrated by the following example.

Example 5.1: m = 2. The jobs are $J_1, J_2, \ldots, J_{n+1}$, where $t_i[1] = t_i[2] = 1,\ 1 \le i \le n$;

$t_{n+1}[1] = x; \quad t_{n+1}[2] = 1; \quad x > n.$

The schedules Z and Z_o are given in Figure 24. In the figure, tasks for the i-th job are indicated by integer i. (In general, an optimal mean flow schedule is not a minimal length schedule. This explains the fact that schedule Z_o in Figure 24 is actually longer than schedule Z.)

$$\begin{aligned} mft(Z) &= (x + 1) + \sum_{i=1}^{n}(x + i) \\ &= (n + 1)x + n(n + 1)/2 + 1. \\ mft(Z_o) &= \sum_{i=1}^{n-1}(i + 1) + n + x + (n + 1) \\ &= x + n(n + 5)/2. \end{aligned}$$

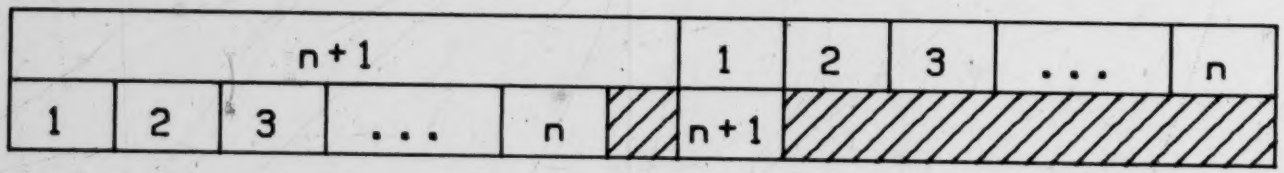
Therefore, $\dfrac{mft(Z)}{mft(Z_o)} = \dfrac{(n + 1)x + n(n+1)/2 + 1}{x + (n + 5)/2}$

which approaches n+1 as x approaches infinity. ∎

Now consider the case for the SPT heuristic, in which jobs are processed in order of non-decreasing processing time. The rule is normally implemented as follows: Suppose that the j-th processor is available, and jobs J_i and J_k have no tasks currently under execution and their tasks for the j-th processor have not yet been executed, then $J_i[j]$ is chosen to execute before $J_k[j]$ if $T_i \le T_k$.

Theorem 5.3: Let Z_o be an optimal mean flow time schedule for an m-processor n-job flow shop. Let Z_s be a schedule constructed with the SPT heuristic.

Schedule Z:



Schedule Z_0:

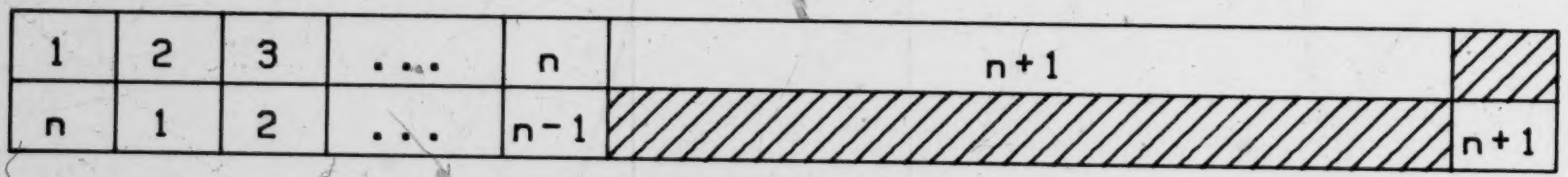


FIGURE 24: Schedules for Example 5.1

Then, $\frac{mft(Z_s)}{mft(Z_o)} \leq m.$

Proof: Assume $T_1 \leq T_2 \leq \ldots \leq T_n$.

$$mft(Z_s) = \Sigma_{i=1}^{n}(F_s(J_i)) \leq \Sigma_{i=1}^{n}(\Sigma_{k=1}^{i}(T_k)).$$

Let $(q(1), q(2),\ldots,q(n))$, a permutation of the first n integers, be the order in which the jobs are completed in schedule Z_o. Then,

$$F_o(J_{q(i)}) \geq \Sigma_{k=1}^{i}(T_{q(k)}/m) \geq \Sigma_{k=1}^{i}(T_k/m).$$

Therefore,

$$mft(Z_o) = \Sigma_{i=1}^{n}(F_o(J_{q(i)})) \geq \Sigma_{i=1}^{n}(\Sigma_{k=1}^{i}(T_k/m))$$
$$\geq mft(Z_s)/m. \quad \square$$

As for the previous theorem, the bound is asymptotically tight as illustrated by Example 5.2.

Example 5.2: There are m+1 jobs, J_1, ... J_{m+1} for m processors, where

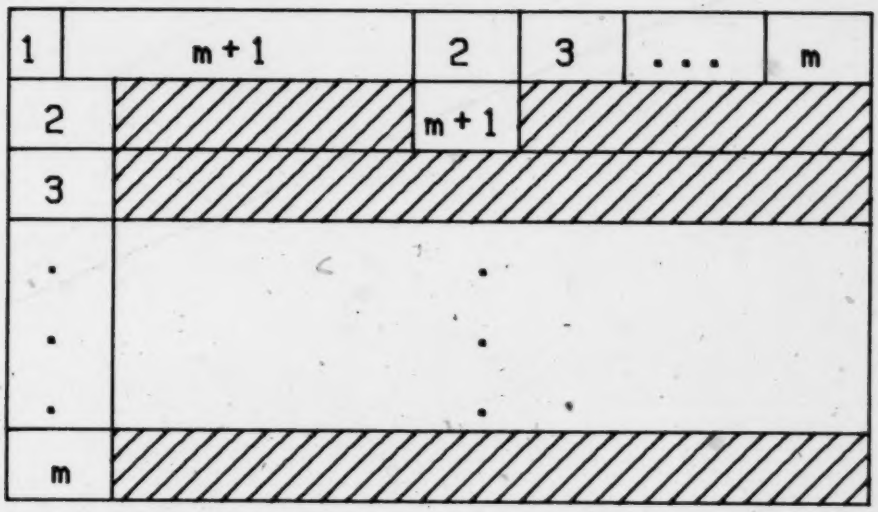
$$t_1[j] = \begin{cases} 1 \text{ for } j = 1. \\ 0 \text{ for } 2 \leq j \leq m. \end{cases}$$

$$t_i[j] \quad 2 \leq i \leq m = \begin{cases} 2 \text{ for } j = 1, \text{ or } j = i, \\ 0 \text{ for } 2 \leq j \leq m,\ j \neq i. \\ x \text{ for } j = 1\ (x > 2). \end{cases}$$

$$t_{m+1}[j] = \begin{cases} 2 \text{ for } j = 2. \\ 0 \text{ for } 3 \leq j \leq m. \end{cases}$$

The SPT and optimal schedules, Z_s, Z_o, are given in Figure 25. As in Example 5.1, tasks for the i-th job are indicated in the figure by integer i.

Schedule Z_1:



Schedule Z_2:

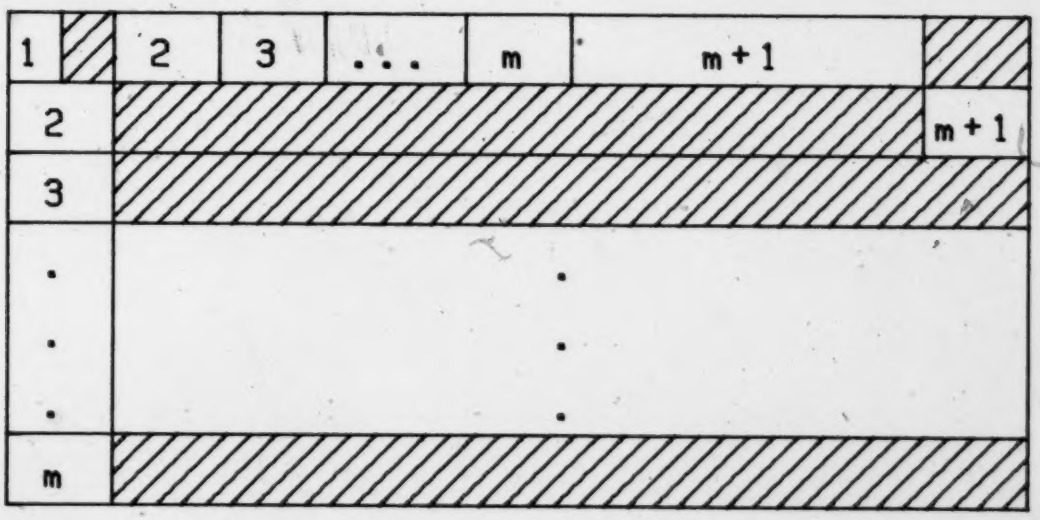


FIGURE 25:
Schedules for Example 5.2

$$mft(Z_s) = 1 + (x + 3) + \sum_{i=1}^{m-1}(x + 1 + 2i)$$
$$= mx + m^2 + 3.$$
$$mft(Z_o) = 1 + 2m + 2 + x + \sum_{i=1}^{m-1}2(i + 1)$$
$$= x + m(m + 3) + 1.$$

Therefore, $\frac{mft(Z_s)}{mft(Z_o)} = \frac{mx + m^2 + 3}{x + m(m + 3) + 1}$

which approaches m as x approaches infinity. ∎

5.4 Discussion

The main contribution of this chapter is the reduction from 3-PARTITION of the 2-processor open shop mean flow time scheduling problem, thus showing the problem to be NP-complete. One may conclude as well that the problem remains NP-complete when the number of processors $m > 2$. Thus the relaxation of the constraint that each job's tasks be processed in the same processor order in the flow shop model, yielding an open shop, still leaves an intrinsically difficult problem.

In addition to the above, tight bounds have been derived for the mean flow time of an arbitrary schedule and for an SPT schedule in terms of the optimal mean flow time. Since the number of jobs is usually much larger than the number of processors, the bounds indicate some advantage of SPT schedules over arbitrary schedules.

Chapter Six

CONCLUDING REMARKS

Deterministic processor scheduling is of practical and theoretical value to computer science as well as many other disciplines and as such its study is a worthwhile endeavour. In this thesis, several models of scheduling theory have been considered, relevant previous work surveyed and some significant results were obtained.

For some minimal length problems, polynomial algorithms have been developed; namely, an $O(n^6)$ algorithm for LMS and SML three-processor flow shops and an $O(n^2)$ algorithm for an m-processor bound UET system with two task chains. An $O(n^2 I)$ algorithm is also given for a tree-structured set of tasks on a 2-processor bound UET system, where I is the number of terminal subsets of the tree. The later algorithm can be extended, with corresponding increase in processing time, to more than two processors. This algorithm is not polynomial in n but is a significant improvement over the alternative of simple enumeration. Furthermore, the dynamic programming technique outlined in the algorithm can be applied with any terminal subset enumerator to provide solutions for similar systems with more complex precedence constraints.

Several problems were also shown to be NP-complete. These are minimizing schedule length on two-procesor bound

UET systems even when the precedence constraints consist of chains only, 2-maximal three-processor flow shops and 1 or 3 maximal/minimal flow shops, and minimizing the mean flow time on the two-processor open shop. In every case, the result trivially holds when the number of processors is increased. With the exception of the 2-maximal flow shop, the results are strong NP-complete reductions from the 3-PARTITION problem.

Finally, in the area of performance bounds, tight bounds were obtained on the lengths of list schedules on identical processors for independent tasks with similar execution times, and on the mean flow times of arbitrary and SPT schedules for the open shop.

There are still many challenging open problems in this field. The distinction between problems which have polynomial solutions and the NP-complete problems is a useful one and there are problems for which this classification is yet to be accomplished. These include minimizing the schedule length for equal execution time tasks for a fixed number of processors $m \geq 3$, and minimizing the mean flow time for m processors, equal execution time tasks and equal weigths, where the precedence relations constitute a tree or forest.

There is also much to be done in the design and analysis of heuristic algorithms for the hard, NP-complete,

problems. The fairly recent technique of approximation algorithms (Sahni, 1975) is virtually untapped and may lead to exciting new solutions.

Although the work presented here has concentrated on the more common models, there are others more suitable and realistic for some applications. These include some types of processor bound systems such as the job shop (Baker, 1974) and typed systems (Liu and Liu, 1977; Jaffe, 1978). These models as well as those considered in this thesis may also be studied in connection with other performance measures, such as minimizing lateness and tardiness, and minimizing the mean number of tasks in the system at any time.

BIBLIOGRAPHY

Achugbue J. O., Chin F. Y. (1978); "Bounds On Schedules For Independent Tasks With Similar Execution Times", (to appear Journal of the ACM)

Achugbue J. O., Chin F. Y. (1979a); "On Optimal Schedules For Processor Bound Systems", Dept. of Computing Science, University of Alberta, August.

Achugbue J. O., Chin F. Y. (1979b); "Complexity and Solutions of Some Three Stage Flow Shop Scheduling Problems", Dept. of Computing Science, University of Alberta, August.

Achugbue J. O., Chin F. Y. (1979c); "Scheduling the Open Shop to Minimize Mean Flow Time", Dept. of Computing Science, University of Alberta, Sept.

Achugbue J. O., Chin F. Y. (1980); "An Algorithm for Terminal Subset Enumeration", Dept. of Computing Science, University of Alberta, (in prep).

Aho A. V., Hopcroft J. E., Ullman J. D. (1974); The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass.

Baker K. R. (1974); Introduction To Sequencing And Scheduling, John Wiley And Sons, New York.

Baker K. R. (1975); "A Comparative Study of Flowshop Algorithms", Operations Research 23(1), Jan-Feb, pp62-73.

Baker K. R. (1975a); "An Elimination Method for the Flowshop Problem", Operations Research 23(1), Jan-Feb, pp159-162.

Baker K. R., Schrage L. E. (1978); "Finding an Optimal Sequence by Dynamic Programming: An Extension To Precedence Related Tasks", Operations Research 26(1), Jan-Feb, pp111-120.

Brown K. Q. (1979); "Dynamic Programming in Computer Science", Technical Report CMU-CS-79-106, Dept. of Computer Science, Carnegie-Mellon University.

Bruno J. E., Coffman Jr. E. G., Sethi R. (1974); "Scheduling Independent Tasks To Reduce Mean Finishing Time", Communications of the ACM 17(7), July, pp382-387.

Bruno J. E., Coffman Jr. E. G., Sethi R. (1974a); "Algorithms for Minimizing Mean Flow Time", Proceedings IFIPS Congress, North Holland Publ., August, pp504-510

Bruno J., Sethi R. (1975); "On the complexity of mean flow time scheduling", Tech Report, Computer Science Dept, Penn State Univ.

Burns F., Rooker J. (1975); "A Special Case of the 3xn Flowshop Problem", Naval Research Logistics Quarterly 22, pp811-817.

Burns F., Rooker J. (1976); "Johnson's Three-Machine Flowshop Conjecture", Operations Research 24(3), pp578-580.

Burns F., Rooker J. (1978); "Three-Stage Flow-Shops With Recessive Second Stage", Operations Research 26(1), Jan-Feb, pp207-208

Coffman Jr. E. G., Graham R. L. (1972); "Optimal Scheduling for Two-Processor Systems", ACTA Informatica 1, pp200-213.

Coffman Jr. E. G., Garey M. R., Johnson D. S. (1978); "An Application of Bin Packing to Multiprocessor Scheduling", SIAM Journal Of Computing 7(1), Feb., pp1-17.

Coffman Jr. E. G. (Ed) (1974); Computer and Job/Shop Scheduling Theory, John Wiley & Sons, New York.

Coffman Jr. E. G., Sethi R. (1976); "Algorithms Minimizing Mean Flow Time Schedule-Length Properties", ACTA Informatica 6(1), pp1-14.

Coffman Jr. E. G., Labetoulle J. (1977); "Flow Time Sequencing of Independent Tasks", INFOR 15(3), October, pp289-307.

Conway R. W., Maxwell W. L., Miller L. W. (1967); Theory Of Scheduling, Addison-Wesley Publ., Reading, Mass.

Cook S. A. (1971); "The Complexity of Theorem-Proving Procedures", Proceedings, 3rd Annual ACM Symposium On Theory Of Computing, pp151-158.

Chin F. Y., Tsai L. (1978); "On J-maximal And J-minimal Flow Shop Schedules", Dept. of Computing Science, University of Alberta, .

Eastman W. L., Even S., Issacs I. M. (1964); "Bounds for the Optimal Scheduling of n Jobs on m Processors," Management Science 11, pp268-278.

Garey M. R. (1973); "Optimal Task Sequencing with Precedence Constraints", Discrete mathematics 4, pp37-56.

Garey M. R., Johnson D. S. (1975); "Complexity Results for Multiprocessor Scheduling under Resource Constraints", SIAM Journal of Computing 4(4), Dec., pp397-411.

Garey M. R., Johnson D. S. (1978); '"Strong" NP-Completeness Results: Motivations, Examples, and Implications", Journal of the ACM 25(3), pp499-508.

Garey M. R., Johnson D. S. (1978a); Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., San Francisco.

Garey M. R., Johnson D. S., Sethi R. (1976); "The Complexity of Flowshop and Jobshop Scheduling", Maths. of Operations Research 1, pp117-129.

Garey M. R., Graham R. L., Johnson D. S. (1978); "Performance Guarantees for Scheduling Algorithms", Operations Research 26(1), Jan-Feb, pp1-21.

Gonzalez M. J.(1977); "Deterministic Processor Scheduling," ACM Computing Surveys 9(3), September, pp173-204.

Gonzalez T. (1976); "A Note on Open Shop Preemptive Schedules", Technical Report #214, Computer Science Dept, Penn State Univ, December.

Gonzalez T. (1979); "NP-hard Shop Problems", Technical Report CS-79-35, Computer Science Dept, Penn State Univ., May.

Gonzalez T., Ibarra O. H., Sahni S. (1977); "Bounds for LPT Schedules on Uniform Processors", SIAM Journal Of Computing 6(1), March, pp155-166.

Gonzalez T., Sahni S. (1976); "Open Shop Scheduling to Minimize Finish Time", Journal of the ACM 23(4), Oct, pp665-679.

Gonzalez T., Sahni S. (1978); "Flowshop and Jobshop Schedules: Complexity and Approximation", Operations Research 26(1), Jan-Feb, pp36-52.

Goyal D. K. (1977); "Scheduling Processor Bound Systems", Proceedings of the Sixth Texas Conference on Computing Systems, pp7B-21 to 7B-25.

Graham R. L. (1966); "Bounds for Certain Multiprocessing Anomalies", Bell System Technical Journal 45, pp1563-1581.

Graham R. L. (1969); "Bounds for Multiprocessing Timing Anomalies", Siam Journal of Applied Maths. 17(2), pp416-429.

Graham R. L. (1972); "Bounds on Multiprocessing Anomalies and Related Packing Algorithms", Proceedings AFIPS Conference 40, pp205-217.

Graham R. L. (1974); "Bounds on the Performance of Scheduling Algorithms", in Computer and Job/Shop Scheduling Theory, ed. E. G. Coffman Jr., John Wiley & Sons, pp165-228.

Gupta J. N. D. (1975); "Analysis of a Combinatorial Approach to Flowshop Scheduling Problems", Operations Research Quarterly 26(2), July, pp431-440.

Held M., Karp R. (1962); "A Dynamic Programming Approach to Sequencing Problems", SIAM Journal of Applied Maths. 10(1), pp196-210.

Hirschberg D. S. (1975); "A Linear Space Algorithm for Computing Maximal Common Subsequences", Communications of the ACM 18, pp341-343.

Horn W. A. (1972); "Single-Machine Job Sequencing with Tree-like Precedence Ordering and Linear Delay Penalties", SIAM Journal of Applied Maths 23, pp189-202.

Horn W. A. (1973); "Minimizing Average Flow Time with Parallel Machines", Operations Research 21, pp846-847.

Hu T. C. (1961); "Parallel Sequencing and Assembly Line Problems", Operations Research 9(6), pp841-848.

Ibarra O. H., Kim C. E. (1975); "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems", Journal of the ACM 22(4), October, pp463-468.

Ibarra O. H., Kim C. E. (1977); "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors", Journal of the ACM 24(2), Apr., pp280-289.

Ignall E., Schrage L. (1965); "Application of the Branch and Bound Technique to Some Flow-Shop Scheduling Problems", Operations Research 13(3), pp400-412.

Jaffe, J. M. (1978); "Bounds on the Scheduling of Typed Task Systems", Technical Report MIT/LCS/TM-111, Massachusetts Institute of Technology, Cambridge, MA.

Johnson S. M. (1954); "Optimal Two-and-Three-Stage Production Schedules", Naval research and Logistics Quarterly 1(1), pp61-68.

Karp R. M. (1972); "Reducibility Among Combinatorial Problems", in Complexity of Computer Computations, R. E. Miller & J. W. Thatcher (Eds), Plenum Press, New York, pp85-104.

Karp R. M. (1975); "On the Computational Complexity of Combinatorial Problems", Networks 5, pp45-68.

Kohler W. H., Steiglitz K. (1971); "Evolutionary Learning of Neighbourhoods for Heuristic Programs and Application to a Sequencing Problem", Proceedings 19th Allerton Conference on Circuit and Systems Theory, October, pp377-389.

Kohler W. H., Steiglitz K. (1974); "Characterization and Theoretical Comparison of Branch and Bound Algorithms for Permutation Problems", Journal of the ACM 21(1), pp140-156.

Lageweg B. J., Lenstra J. K., Rinnooy Kan A. H. G. (1978); "A General Bounding Scheme for the Permutation Flowshop Problem", Operations Research 26(1), Jan-Feb, pp53-67.

Lawler E. L. (1976); Combinatorial Optimization: Networks and Matroids, Holt, Reinhart & Winston, New York.

Lenstra J. K., Rinnooy Kan A. H. G. (1978); "Complexity of Scheduling Under Precedence Constraints", Operations Research 26(1), Jan-Feb, pp22-35.

Liu J. W. S., Liu C. L. (1977); "Performance Analysis of Multiprocessor Systems Containing Functional Dedicated Processors", Technical Report UIUCDCS-R-77-835, Dept. of Computer Science, University of Illinois at Urbana-Champagne.

McNaughton R. (1959); "Scheduling with Deadlines and Loss Functions", Management Science 6(1), October.

Muntz R. R., Coffman Jr. E. G. (1969); "Optimal Preemptive Scheduling on Two-Processor Systems", IEEE Transactions On Computers C-18(11), pp1014-1020.

Muntz R. R., Coffman Jr. E. G. (1970); "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems", Journal of the ACM 17(2), pp324-338.

Nijenhuis A., Wilf H. S. (1978); Combinatorial Algorithms for Computers and Calculators, Second Edition, Academic Press.

Sahni S. (1976); "Algorithms for Scheduling Independent Tasks", Journal of the ACM 23(1), pp116-127.

Sahni S. (1975); "Approximate Algorithms for the 0/1 Knapsack Problem", Journal of the ACM 22(1), January, pp115-124.

Smith W. E. (1956); "Various Optimizers for Single Stage Production", Naval Research Logistics Quarterly 3(1).

Sethi R. (1977); "On the Complexity of Mean Flow Time Scheduling", Maths. of Operations Research 2(4), November, pp320-330.

Swarc W. (1977); "Optimal Two-Machine Orderings in the 3xn Flow-Shop Problem", Operations Research 25(1), pp70-77.

Ullman J. D. (1973); "Polynomial Complete Scheduling Problems", Operating Systems Review 7(4), pp96-101.

Ullman J. D. (1974); "Complexity of Sequencing Problems", in Computer & Job-Shop Scheduling Theory, E. G. Coffman Jr. (Ed), John Wiley & Sons, pp139-164.

Ullman J. D. (1975); "NP-Complete Scheduling Problems", Journal of Computer & Systems Science 10(3), June, pp384-393.

Weide B. (1977); "A Survey of Analysis Techniques for Discrete Algorithms", ACM Computing Surveys 9(4), December, pp291-313.